

**Heathkit**



**Educational  
Systems**

# **ROBOTICS AND INDUSTRIAL ELECTRONICS**

Model EE-1800  
HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022  
595-2726

**Copyright © 1982**  
First Printing  
Heath Company  
Not Affiliated with D.C. Heath  
*All Rights Reserved*  
Printed in the United States of America

## CONTENTS

INTRODUCTION .....	3
COURSE OBJECTIVES .....	5
COURSE OUTLINE .....	6
PARTS LIST .....	23
UNIT 1 — ROBOT FUNDAMENTALS .....	1-1
UNIT 2 — AC AND FLUIDIC POWER .....	2-1
UNIT 3 — DC POWER AND POSITIONING .....	3-1
UNIT 4 — MICROPROCESSOR FUNDAMENTALS .....	4-1
UNIT 5 — INTRODUCTION TO PROGRAMMING .....	5-1
UNIT 6 — A TYPICAL MICROPROCESSOR CONTROLLER .....	6-1
UNIT 7 — DATA ACQUISITION .....	7-1
UNIT 8 — DATA HANDLING AND CONVERSION .....	8-1
UNIT 9 — VOICE SYNTHESIS .....	9-1
UNIT 10 — ET-18 INTERFACING .....	10-1
UNIT 11 — INDUSTRIAL ROBOTS AT WORK .....	11-1
UNIT 12 — EXPERIMENTS .....	12-1
APPENDIX A — NUMBER SYSTEMS AND CODES .....	A-1
APPENDIX B — 6808 DATA SHEETS .....	B-1
APPENDIX C — DEFINITION OF THE EXECUTABLE INSTRUCTIONS .....	C-1
APPENDIX D — THE PHONEME DICTIONARY .....	D-1



## INTRODUCTION

ROBOT — a word that is becoming increasingly popular — has different meanings for different people. To some, it signifies R2D2 and C-3PO, the “human-like” androids of “Star Wars” fame. To others, it defines the intelligent machine of the future, dedicated to serving man. It is the intent of this course to view the robot as an intelligent machine; except, it is not a device of the future. It is here today!

Evolving in the early 1960's, industrial robots experienced limited growth during their first decade of existence. At first, these robots were no more than extensions of automated machines. Within the last few years, however, these devices have shown an almost phenomenal rate of growth in both numbers and diversity of uses. This rapid growth can be mainly attributed to the technological advancements made in the field of microprocessors and microcomputers, the brains of the sophisticated robot. With the development of larger, more powerful microcomputers, thus increasing the capabilities of the robot even more, an even brighter future is predicted for these intelligent devices.

All the electronic concepts associated with robots and robotics are not new. Many of the basic principles were discovered years ago. However, the microcomputer has provided a method by which these principles can easily be jointed together and controlled. In this course, you will review some of the basic concepts, learn new concepts associated with robotics, and finally, through the use of a microprocessor controller, join the old and the new to develop a working robot.

In addition, this course will provide you with the basic knowledge to keep abreast of this swiftly evolving, diversified field.

## How to Use This Course

How do you gauge your learning? Let the **objectives** be your guide. These carefully constructed objectives are the framework for the course. When you meet all of the objectives, you will have satisfied the requirements of the course. You will find two types of objectives in this course: Broad, **Course Objectives** are listed following the introduction. More specific **Unit Objectives** are listed near the front of each unit. When you can satisfy these unit objectives, you have learned everything that was intended from the units; no matter how easy it seemed.

Don't neglect the appendices to this course. They were designed to help you evaluate and use your newly acquired knowledge. Also, during your studies, you will encounter many **Programmed Reviews** which are designed to reinforce the material presented. We suggest you use these reviews as short quizzes to test your understanding of the material. Each **Unit Examination** is supplied with the correct answers to further enhance your learning experience. The Programmed Reviews and Unit Examinations can be used as a guide to determine areas that may require further study.

To perform the experiments in this learning program, you will need an ET-18 Robot Trainer, a multimeter, and a few common hand tools. All other necessary parts such as ICs, switches, resistors, hookup wire, etc. are furnished. Be sure to check your parts against the parts list. If you are missing any, you must request them on the order sheet provided.

## COURSE OBJECTIVES

This course is designed to provide you with:

1. An understanding of the terminology associated with low, medium, and high-technology industrial robots.
2. A knowledge of the operation of various AC and DC motors and generators.
3. A basic insight into the operation of industrial hydraulic and pneumatic systems.
4. In-depth information on portable power sources.
5. An understanding of the terminology and basic design of microprocessors and microcomputers.
6. The knowledge required to write robot control programs.
7. A thorough understanding of a typical microprocessor controller.
8. A knowledge of the types of sensing systems required, to obtain data, in order for a robot to perform specific tasks.
9. An understanding of how analog data is changed into digital data for use by the microprocessor.
10. Information on how to make a robot talk, using voice synthesis.
11. Insight into "open and closed-loop" control systems.
12. In-depth information on RAM, ROM, and I/O interfacing.
13. A knowledge of the types of programming methods used to teach industrial robots.
14. An understanding of why external sensors play such an important role in the operation of industrial robots.
15. The programming skills required to make the ET-18 Robot Trainer perform a variety of tasks.

## COURSE OUTLINE

### UNIT 1     **Robot Fundamentals**

Introduction

Unit Objectives

Unit Activity Guide

Robot Evolution

Terminology

    Axes

    Manipulator

    Actuators

    Gripper

    Controller

    Power Supply

    Cylindrical Coordinate Robot

    Spherical Coordinate Robot

    Jointed-Spherical Coordinate Robot

    Non-Servo Robots

    Servo-Controlled Robots

        Point-to-Point Servo Control

        Continuous Path Servo Control

Low-Technology Robots

    Characteristics

        Degrees of Freedom

        Payload

        Speed

        Accuracy

        Actuation

        Controllers

    Operation

Medium and High Technology Robots

    The Medium-Technology Robot

        Axes

        Payload

        Speed

        Accuracy

        Actuation

        Controller

    The High-Technology Robot

        Puma Family

            250 Series

            500 Series

            600 Series

General Characteristics  
Other High-Technology Robots  
Unit Examination  
Examination Answers

## **UNIT 2      AC and Fluidic Power**

Introduction

Unit Objectives

Unit Activity Guide

AC Power Generation

AC Generator Characteristics

Types of AC Generators

Revolving Armature Type Generator

Revolving Field Type Generator

Rating of AC Generators

Operation of a Basic AC Generator

Single Phase Generators

Three-Phase Generators

The WYE Connection

The DELTA Connection

AC Generator Considerations

Single and Three-Phase Voltage Outputs

Regulation of AC Generators

AC Motors

Rotating Field

Three-Phase Induction Motor

Induction Motor Stator

Induction Motor Rotor

Cage Rotor

Form-Wound Rotor

Induction Motor Slip

Synchronous Motor

Principle of Operation

Starting a Synchronous Motor

Single-Phase Motors

Split-Phase Motor

Capacitor Motor

Basic Hydraulic System

Oil Reservoir

Hydraulic Pumps

Vane Pump

Hydraulic Valves

- Directional Control Valves
  - One-Way Valves
  - Two-Way Valves
    - Globe Valve
    - Gate Valve
    - Spool Valve
  - Three-Way Valve
  - Four-Way Valve
- Flow Control Valves
- Pressure Control Valves
- Hydraulic Actuators
  - Single-Acting Actuator
  - Double-Acting Actuator
  - Rotary Actuator
- Pneumatic Systems
  - Pneumatic System Components
    - Air Compressor/Motor/Air Storage Tank Unit
  - Shut-Off Valves
  - Processing and Conditioning Units
  - Control Valves and Actuators
- Unit Examination
- Examination Answers

**UNIT 3     DC Power And Positioning**

- Introduction
- Unit Objectives
- Unit Activity Guide
- Batteries (Overview)
  - Cell
  - Electrodes
  - Electrolyte
  - Primary Cell
  - Secondary Cell
  - Battery
- Nickel-Cadmium Batteries
  - Construction
  - Operation of Nickel-Cadmium Cells
    - Reversal Protection
  - General Characteristics
    - Charge Retention
    - Capacity
    - Rated Capacity
  - Charging and Charging Techniques
    - Trickle Charge
    - Minimum Charge



- Standard Charge
- Quick Charge
- Rapid Charge
- Changing Parameters
- Alternate Charging Circuits
- Care and Maintenance of Nickel-Cadmium Batteries
  - Prior to Use
  - Circuit Connection
  - Nickel-Cadmium Use
  - Charging
  - General
  - Reversible Failures
  - Permanent Failures
- Gelled-Electrolyte Batteries
  - Construction
  - Gelled-Electrolyte Parameters
    - Discharge Operating Temperature
  - Storage
  - Cell Reversal
  - Battery Life
- Capacity
  - Capacity Performance Nomogram
- Gelled-Electrolyte Batteries vs Pulse Applications
- Charging and Charging Techniques
  - Charging Principles
  - Charger Types
    - Ideal (Multimode) Charger
    - Constant Voltage Chargers
    - Float Charger
    - Constant Current Charger
- Charging Considerations
  - Initial Charge Current
  - End of Charge Current
  - Temperature Effects
- General Information
  - Circuit Connection
  - Series and Parallel Use
  - Charging
  - Battery Handling
  - Battery Failures
    - Reversible Failures
    - Permanent Failures
- Nickel-Cadmium vs Gelled-Electrolyte
- DC Motors
  - Characteristics of DC Motors
  - Components of a DC Motor

Field Structure	
Field Coils	
Armature	
Brush Assembly	
Torque	
Speed Regulation	
Speed vs Torque	
Types of DC Motors	
Shunt-Wound	
Series-Wound	
Compound-Wound	
Speed Control of Wound-Field DC Motors	
Shunt-Field Control	
Armature-Voltage Control	
Motor Selection Factors	
Speed Range	
Speed Control Under Load	
Reversing a DC Motor	
DC Brushless Motors	
Transistor Switched Brushless DC Motors	
Hall-Effect Motor	
Operation of the Hall-Effect Motor	
The Hall Generator	
Stepper Motors	
Stepping Motor Defined	
Stepping Motor Types	
Bipolar Permanent Magnet Stepper	
Permanent Magnet Stepper Motor Operation	
Variable Reluctance Stepper Motor	
Bifilar Stepping Motor	
Stepper Motor Control	
Bipolar Control	
Unipolar Control	
Unit Examination	
Examination Answers	



## **UNIT 4     Microprocessor Fundamentals**

Introduction

Unit Objectives

Unit Activity Guide

Terms and Conventions

    Microprocessor Defined

    Microcomputer Defined

    Input/Output Devices

    Port Defined

    Stored Program Concept

    Computer Words

    Word Length

An Elementary Microcomputer

    The Microprocessor Unit

        Arithmetic Logic Unit

        Accumulator

        Data Register

        Address Register

        Program Counter

        Instruction Decoder

        Controller-Sequencer

    Memory

        Read Signal

        Write Signal

        RAM

        ROM

    Fetch-Execute Sequence

    A Sample Program

Executing a Program

    The Fetch Phase

    The Execute Phase

    Fetching an ADD Instruction

    Executing an ADD Instruction

    Fetching and Executing the HLT Instruction

Addressing Modes

    Inherent or Implied Addressing

    Immediate Addressing

    Direct Addressing

    Sample Program Using Direct Addressing

    Executing the Sample Program

    Combining Addressing Modes

Binary Arithmetic

    Binary Addition

    Binary Subtraction

- Binary Multiplication
- Binary Division
- Representing Negative Numbers
  - Sign and Magnitude
  - One's Complement
  - Two's Complement
- Two's Complement Arithmetic
- Ten's Complement Arithmetic
- Two's Complement Subtraction
- Arithmetic With Signed Numbers
  - Adding Positive Numbers
  - Adding Positive and Negative Numbers
  - Adding Negative Numbers
- Boolean Operations
  - AND Operation
  - OR Operation
  - Exclusive OR Operation
  - Invert Operation
- Unit Examination
- Examination Answers

## **UNIT 5     Introduction To Programming**

- Introduction
- Unit Objectives
- Unit Activity Guide
- Branching
  - Relative Addressing
  - Executing a Branch Instruction
  - Branching Forward
  - Branching Backward
- Conditional Branching
  - Condition Codes
    - Negative (N) Register
    - Zero (Z) Register
    - Carry (C) Register
    - Overflow (V) Register
  - Conditional Branch Instructions
- Algorithms
  - Multiplying by Repeated Addition

Dividing by Repeated Subtraction

Converting BCD to Binary

Converting Binary to BCD

Additional Instructions

Add with Carry Instruction (ADC)

Subtract with Carry Instruction (SBC)

Arithmetic Shift Accumulator Left Instruction (ASLA)

Decimal Adjust Accumulator Instruction (DAA)

Unit Examination

Examination Answers

## **UNIT 6     A Typical Microprocessor Controller**

Introduction

Unit Objectives

Unit Activity Guide

Architecture of the 6808 MPU

Programming Model of the 6808 MPU

Two Accumulator

16-Bit Program Counter

Condition Code Registers

Index Register

Stack Pointer

Block Diagram of the 6808 MPU

Instruction Set of the 6808 MPU

Arithmetic Instructions

Data Handling Instructions

Logic Instructions

Data Test Instructions

Index Register and Stack Pointer Instructions

Branch Instructions

Condition Code Register Instructions

New Addressing Modes

Extended Addressing

Indexed Addressing

Purpose

Instruction Format

Determining the Operand Address

Adding a List of Numbers

Copying a List

Instruction Set Summary

Stack Operations

Cascade Stack

The PUSH Instruction

- The PULL Instruction
- Memory Stack
  - The Stack Pointer
  - The PUSH Instruction
  - The PULL Instruction
  - Using the Stack
- Subroutines
  - Jump Instruction (JMP)
  - Jump to Subroutine Instruction (JSR)
  - Return From Subroutine Instruction (RTS)
  - Nested Subroutines
  - Branch to Subroutine Instruction (BSR)
  - Summary of Subroutine Instructions
- Input/Output Operations (I/O)
  - Output Operations
  - Input Operations
  - Input/Output Programming
  - Program Control of I/O Operations
  - Interrupt Control of I/O Operations
- Interrupts
  - RESET
  - Non-Maskable Interrupts (NMI)
  - Return from Interrupt Instruction (RTI)
  - Interrupt Request (IRQ)
  - Interrupt Mask Instructions
    - Set-Interrupt-Mask Instruction (SEI)
    - Clear-Interrupt-Mask Instruction (CLI)
  - Software Interrupt Instruction (SWI)
  - Wait for Interrupt Instruction (WAI)
- Unit Examination
- Examination Answers

**UNIT 7     Data Acquisition**

Introduction

Unit Objectives

Unit Activity Guide

Optoelectronic Sensors

    Light-Emitting Diodes

        LED Operation

        LED Construction

    Phototransistors

        Phototransistor Construction

        Phototransistor Operation

    Optoelectronic Sensing System Operation

    Interrupter/Reflector Modules

        Interrupter Module

        Reflector Module

Temperature Sensing

    Bimetallic Sensors

    Resistive Sensors

        Resistive Temperature Detector (RTD)

        Thin Film Detectors (TFDs)

    Semiconductor Sensors

Ultrasonic Sensors

    Properties of Sound

    Ultrasonic Sensing Systems

        Piezoelectric Transducers

            Piezoelectric Effect

            Resonant Frequency

    Ultrasonic Thickness Measuring System

    Ultrasonic Detection and Ranging Navigation

        Ultrasonic Detection System Operation

Unit Examination

Examination Answers

**UNIT 8      Data Handling And Conversion**

Introduction

Unit Objectives

Unit Activity Guide

Synchros and Servomechanisms Overview

Synchro System

Synchro Characteristics

Synchro Characteristics (Basic)

Synchro Transmitter (TX)

Synchro Receiver (TR)

Differential Synchro Transmitter (TDX)

Differential Synchro Receiver (TDR)

Control Transformer (CT)

Synchro System Operation

Transmitter and Receiver Synchro System

Synchro Transmitter-Receiver Operation

        Synchro Transmitter-Differential Transmitter-  
        Receiver System        Synchro Transmitter-Differential Receiver-  
        Transmitter System

Control Transformer (CT)

Control Transformer Characteristics

        Synchro Transmitter-Control Transformer  
        Operation

Servomechanisms

Basic Servomechanism (Open-Loop) Operation

Closed-Loop Servomechanism Operation

Digital-to-Analog Converters

General DAC Concepts

Resolution

Settling Time

Accuracy

Types of DACs

DAC Interfacing

DAC Applications

Waveform Generation

Programmable Gain Amplifier and Attenuator

Motor Control and Positioning

ADCs

Analog-to-Digital Converters

General ADC Concepts and Conversion Techniques

Ramp Conversion

Successive Approximation

Integration

---

Interfacing to ADC Devices  
Unit Examination  
Examination Answers

<b>UNIT 9</b>	<b>Voice Synthesis</b>
	Introduction
	Unit Objectives
	Unit Activity Guide
	How Do We Speak
	The Larynx
	The Vocal Tract
	The Articulators
	Voiceless Sounds
	Fricatives
	Sibilants
	Stops
	Stringing Sounds Together
	Coarticulation
	Fundamentals of Speech
	Naming Sounds
	Phones
	Phonemes and Allophones
	Looking at Waveforms
	What You See
	Breaking Down the Frequency Domain
	Formants
	Producing a Phoneme Electronically
	The Synthesizer Circuit
	Determining Phonemes
	Phoneme Strings
	Phonemes and Voice Inflection
	Phoneme Codes Versus Phoneme Symbols
	Using a Phoneme Synthesizer
	Interconnections
	Supply Voltage
	Timing Sources
	Control Signals
	Strobe Signal
	Acknowledge/Request Signal
	Inflection Code
	Phoneme Data
	Output Amplification

Programming the PSS

Speaking Using Canned Phrases

Programming Unlimited Speech

Inflection Enhancement

Naturalness and Phoneme Phrases

The Phoneme Concatenation Process

The First Step: The Basic Phoneme String

The Second Step: Know Your Options

Pauses

Intonation

The Third Step: Modifying Timing and Intonation

Word Deemphasis

Modifying Intonation

Unit Examination

Examination Answers



**UNIT 10 ET-18 Interfacing**

Introduction

Unit Objectives

Unit Activity Guide

Interfacing Fundamentals

Buses

3-State Logic

The MPU Interface Lines

Instruction Timing

Timing of Program Segment

The 6808 Data Sheets

Interfacing With Random Access Memory

The Static RAM Storage Cell

A 128-Word by 8-Bit RAM

2048-Word by 8-Bit RAM

Connecting RAM to the MPU

Interfacing With Read Only Memory

Mask-Programmed ROM

Programmable ROM or PROM

Erasable Programmable ROM or EPROM

Electrically Erasable ROM or EEROM

The MCM68A364 Mask-Programmed ROM

Connecting ROM to the MPU

Interfacing With Control Circuits

Outputting Data to the Experimental Board

Inputting Data from the Sense Circuit

Unit Examination

Examination Answers

**UNIT 11 Industrial Robots At Work**

Introduction

Unit Objectives

Unit Activity Guide

Industrial Robot Classification

Limited Sequence Robots

Basic Operation

Programmable Robots — With Point-to-Point Control

Basic Operation Using Point-to-Point Control

Programmable Robots — With Continuous Path Control

Application Considerations

Manipulator Configuration

Wrist Assembly

End Effectors

Workpiece Placement

Interfacing to the Task

Interfacing Examples

Unit Examination

Examination Answers

**UNIT 12 Experiments**

## Introduction

## Tools and Equipment

## Format For The Experiments

- Experiment 1: Robot Familiarization — Platform Mobility
- Experiment 2: Robot Familiarization — Manipulator Axes of Motion
- Experiment 3: On-Board Logic Test Probe
- Experiment 4: Manual Control of a DC Motor
- Experiment 5: Straight Line Programs
- Experiment 6: Arithmetic and Logic Instructions
- Experiment 7: Program Branches
- Experiment 8: Additional Instruction
- Experiment 9: New Addressing Modes
- Experiment 10: Arithmetic Operations
- Experiment 11: Stack Operations
- Experiment 12: Subroutines
- Experiment 13: Sensors
- Experiment 14: "Open-Loop" Control of a DC Stepping Motor
- Experiment 15: Robot Voice Routine
- Experiment 16: Robot Language — Motors
- Experiment 17: Robot Language — Sensors and Sound
- Experiment 18: Cassette and Teaching Pendant Interface
- Experiment 19: Modifying a Taught Program

## **APPENDICES**

- Appendix A: Number Systems And Codes
- Appendix B: 6808 Data Sheets
- Appendix C: Definition of the Executable Instructions
- Appendix D: The Phoneme Dictionary

## **FINAL EXAMINATION**

## PARTS LIST

This parts list contains all of the parts for the experiments that you will perform. The key numbers correspond to the numbers on the illustrations. Some of the parts are packaged in envelopes. Except for this initial parts check, keep these parts in their envelopes until they are called for in an experiment.

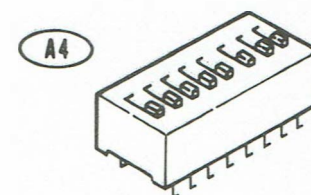
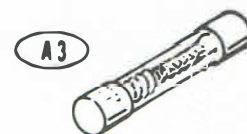
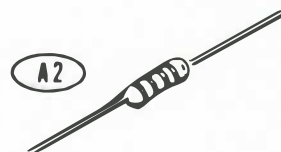
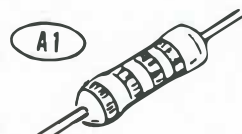
KEY No.	PART No.	QTY.	DESCRIPTION
<b>RESISTORS</b>			
A1	6-150-1	4	15 $\Omega$ , 5%, 1-watt (brown, green, black)
A2	6-104-12	1	100 k $\Omega$ , 5%, 1/4-watt (brown, black, yellow)
A2	6-105-12	1	1 M $\Omega$ , 5%, 1/4-watt (brown, black, green)
A2	6-154-12	2	150 k $\Omega$ , 5%, 1/4-watt (brown, green, yellow)
A2	6-225-12	1	2.2 M $\Omega$ , 5%, 1/4-watt (red, red, green)
A2	6-271-12	2	270 $\Omega$ , 5%, 1/4-watt (red, violet, brown)
A2	6-332-12	4	3300 $\Omega$ , 5%, 1/4-watt (orange, orange, red)
A2	6-333-12	1	3.3 k $\Omega$ , 5%, 1/4-watt (orange, orange, orange)

### FUSES (to be used as necessary)

A3	421-3	1	2 A, 250 V, slow-blow
A3	421-5	1	4 A, 250 V, slow-blow
A3	421-6	1	3 A, 250 V, slow-blow

### SWITCHES

A4	60-653	1	SPST, 8-section DIP
----	--------	---	---------------------



KEY No.	PART No.	QTY.	DESCRIPTION
------------	-------------	------	-------------

**LED's**

A5	412-640	1	2.5 V, 20mA, red
A6	412-642	1	2.5 V, 20mA, green

**CONNECTORS**

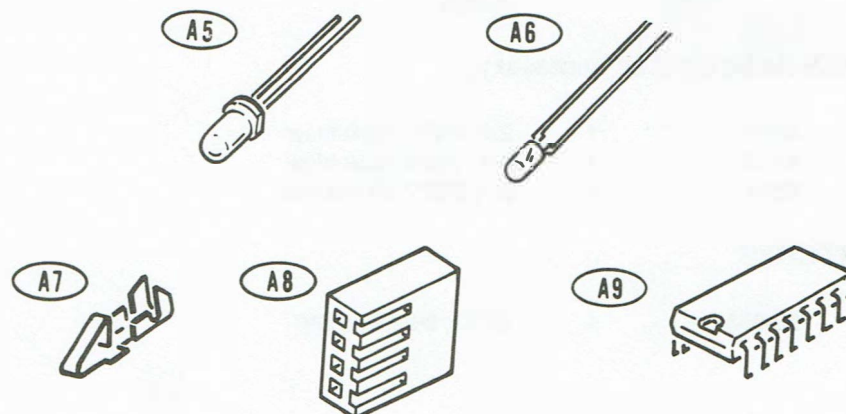
A7	432-753	2	Large spring connectors
A8	432-954	1	4-Pin female connector

**INTEGRATED CIRCUIT (IC)**

A9	442-616	1	LM2901, voltage comparator
----	---------	---	----------------------------

**MISCELLANEOUS**

331-6	1	Solder package
344-59	15'	White #22 wire
347-55	3'	Flat, #24, 8-conductor cable



*Unit 1*

**ROBOT FUNDAMENTALS**

## CONTENTS

Introduction .....	1-3
Unit Objectives .....	1-4
Unit Activity Guide .....	1-5
Robot Evolution .....	1-6
Terminology .....	1-9
Low-Technology Robots .....	1-16
Medium And High Technology Robots .....	1-33
Experiments .....	1-48
Unit Examination .....	1-49
Examination Answers .....	1-53



## INTRODUCTION

To understand robots, both industrial and hobby-type, you need a basic knowledge of automation and robotics principles. A majority of these principles are in the electronics area, but subjects such as basic robot design and classification have to be understood before you proceed. This first unit, therefore, establishes a common background of robotics principles upon which to develop the remainder of the course.

In this unit, you will learn how industrial robots are categorized by their mechanical and electronic capabilities, as well as the types of tasks they are required to perform. You will also be introduced to specific robotics terms that are used throughout this course.

The “Unit Objectives” on the next page state the goals of this unit. Review this list now and again after finishing this unit. Be sure you can satisfactorily complete all the objectives before you take the “Unit Exam.”

The “Unit Activity Guide” follows the “Unit Objectives.” It lists the order in which you should complete this unit.

## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. State the difference between hard automation devices and robots.
2. Define: axes, actuators, manipulators, and controllers.
3. State the difference between servo-controlled and non-servo-controlled robots.
4. List the three basic methods of actuating robots.
5. Identify the most common method of controlling high-technology robots.
6. State why pneumatics is the preferred method of actuating low-technology robots.
7. Define the term “power supply” as it applies to robotics.
8. State which technology robot is best suited for the following tasks:
  - Spray Painting
  - Welding Operations
  - Material Transfer (Light Loads)
  - Machine Loading/Unloading (Medium Loads)
9. Identify the device that made special purpose robots practical.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "Robot Evolution."	_____
<input type="checkbox"/> Answer Programmed Review Questions 1-8.	_____
<input type="checkbox"/> Read "Terminology."	_____
<input type="checkbox"/> Answer Programmed Review Questions 9-18.	_____
<input type="checkbox"/> Read "Low-Technology Robots."	_____
<input type="checkbox"/> Answer Programmed Review Questions 19-29.	_____
<input type="checkbox"/> Read "Medium And High Technology Robots."	_____
<input type="checkbox"/> Answer Programmed Review Questions 30-39.	_____
<input type="checkbox"/> Perform Robot Familiarization Experiments 1 and 2.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## ROBOT EVOLUTION

Before proceeding into the fascinating world of robots, let's take a few minutes and explore robot evolution. Man has been intrigued with mechanisms and "man-like" machines almost since the beginning of time. You can see this in the ancient Greek epic "Iliad" (book 18) where the author, Homer, describes Hephaestus, the God of all mechanical arts, as having two female statues of pure gold which assisted and accompanied him wherever he went. A much later attempt at robot construction is shown in Figure 1-1. This device was described as a "walking locomotive", and was built by George Moore in 1893. It was powered by a 0.5 horsepower gas-fired boiler and could reach a walking speed of 9 mph. The cigar was used as a steam vent for the boiler!

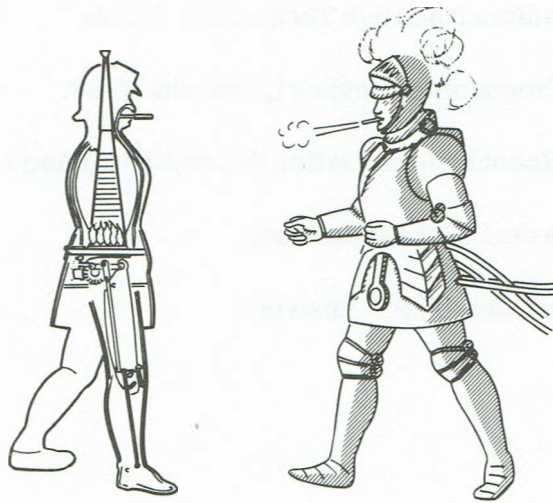


Figure 1-1  
Walking locomotive, 1893.

Man's fascination with robots has greatly increased in the 20th century, as you can see by the wide coverage given them in movies, books, and on television. These robots, while amusing, are not the real robots actually being used today. Today's Industrial Robot is not a walking, flashing, beeping tin can with a monotone voice from a Sci-fi TV series. Rather, it is a pillar-mounted jointed arm that moves, mechanically, on command from a computer.

Industrial robots have emerged from "hard automation", which can be described as equipment designed to accomplish repetitive production-line tasks at a fast pace. A hard automation device consists primarily of drive mechanisms controlled by devices such as timers, cams, switches, and mechanical and electrical stops. The function of hard automation is usually very specific, and there is seldom a need to alter its operation. When changes are required, it is often only a mechanical adjustment. Because of this, there is usually no capability within the equipment to respond to a deviation from the manufacturing process, except perhaps automatic shut-down.

The evolution from hard automation to robots has been in a series of development stages. When speed controllers, sequencers, adjustable stops, timers, and other devices which permitted variations in its motions were added to these hard automations, they became the forefathers of today's robot. The word "ROBOT" was chosen because the device had a small degree of flexibility.

The next step in robot evolution was the addition of servo controls. These feedback devices generated position signals. When the robot moved, these position signals were compared to the original input positioning signals and, once these two signals coincided, the robot ceased movement.

The next and greatest advancement was the adaptation of computer control. The capability to react to externally generated signals was introduced, causing the casual observer to think the robot possessed human intelligence. As the robot continued to grow more sophisticated, its cost also grew. This deluxe, general-purpose robot, equipped with the latest in computer control technology and capable of all sorts of motions, was an overkill in many applications. This brought about the special purpose robots, still with a high degree of sophistication but with less capability and cost than the general purpose robot. This change was made possible by the rapid development of the microprocessor for robot control. Thus, low cost microprocessor controllers made these special-purpose robots a still greater asset to the manufacturing and industrial community.

## Programmed Review

The “programmed reviews” in this course are designed to reinforce your knowledge of the material you study. Read each frame and fill in the blanks. The correct answer is in parentheses at the beginning of the next frame. For best results, use a sheet of paper to cover all frames below the one you are reading.

1. The real robots of today are the _____ robots.
2. (industrial) Industrial robots have emerged from _____.
3. (hard automation) Hard automation devices are _____ altered to change their operation. (seldom/frequently)
4. (seldom) When a change to a hard automation device is required, it is usually a _____ adjustment.
5. (mechanical) Servo controls are _____ devices that generate position signals.
6. (feedback) Servo control systems _____ the input signal to the output signal.
7. (compare) The greatest advancement in the evolution of the robot was the adaptation of _____ control.
8. (computer) Because of their low cost, _____ controllers brought about the rapid development of special purpose robots.
(microprocessor)



## TERMINOLOGY

There are three distinct categories of industrial robots. The basic terms associated with each are similar. Therefore, before discussing each category in detail, we will establish these common terms.

**AXES** — The number of intricate motions a robot can perform is determined by the number of axes the robot has. These axes are also known as “degrees of freedom”. Figure 1-2 shows the typical degrees of freedom most commonly used. The complexity of the task a robot can perform is determined by the number of axes the robot has.

		TECHNOLOGY		
		LOW	MEDIUM	HIGH
D E G R E E  O F  F R E E D O M	Horizontal Travel			
	Vertical Travel	✓		
	Column Rotate	✓	✓	✓
	Waist Bend			
	Shoulder Bend or Pitch		✓	✓
	Elbow Bend			✓
	Extend & Retract	✓	✓	
	Wrist Bend		✓	✓
	Wrist Rotate	✓	✓	✓
	Yaw			✓

Figure 1-2  
Typical degrees of freedom (axes).

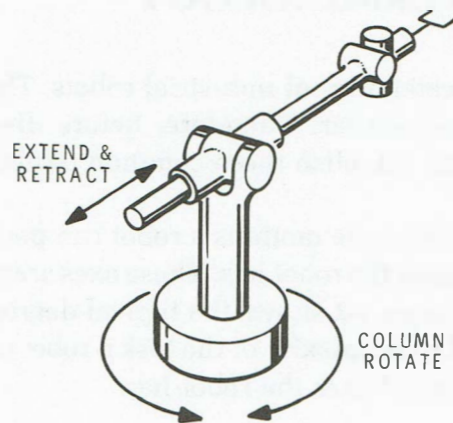


Figure 1-3  
Transfer operation requiring  
two axes of motion.

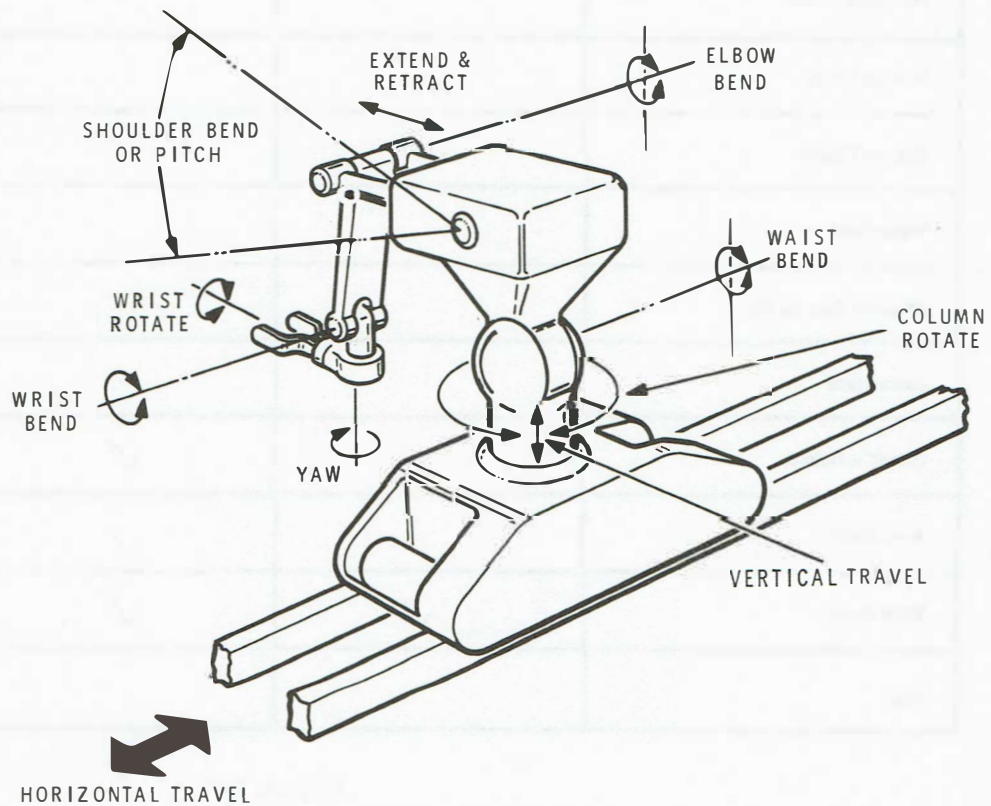


Figure 1-4  
Robot using ten axes of motion.



A robot used for simple transfer operations, as shown in Figure 1-3, may require only two axes such as column rotate and extend/retract. Figure 1-4 depicts a robot that has the capability to follow an automobile down an assembly line, reach inside of the automobile to weld some recessed point, and return to its work station to await the arrival of the next automobile. A robot with all this capability may require up to ten axes of motion. Figure 1-5 shows a typical five-axes robot. Most industrial robot applications require only three to five degrees of freedom to perform a specific task.

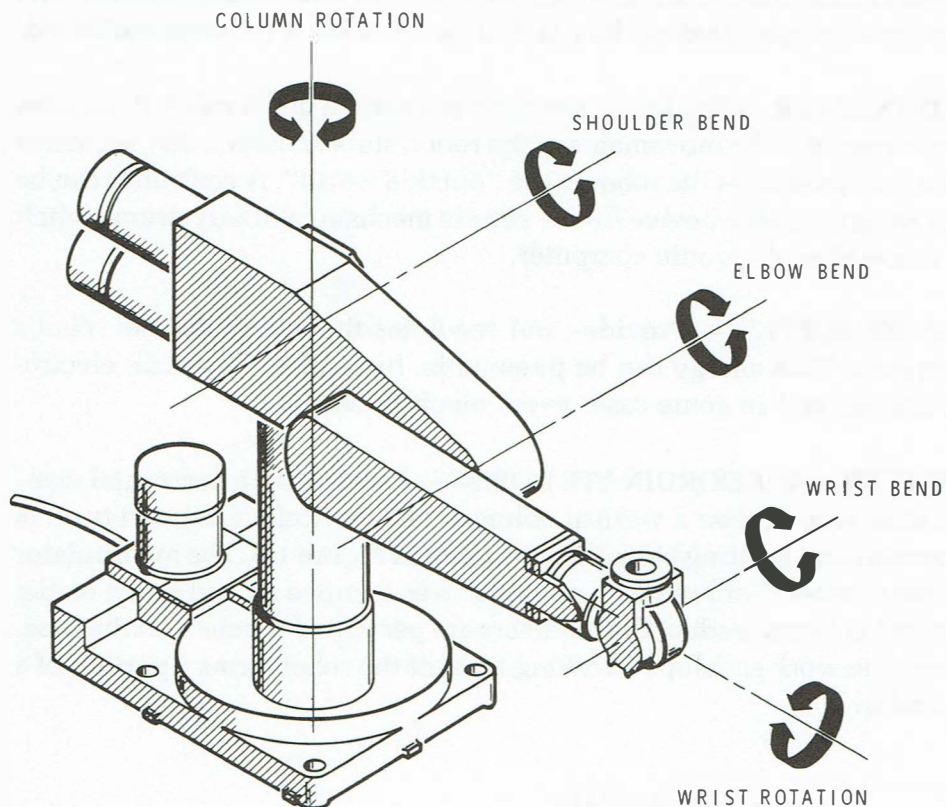


Figure 1-5  
Typical five-axes robot.

**MANIPULATOR** — A system of mechanical linkages and joints that can be moved in various directions to perform the work of the robot. In some instances the manipulator is also called the robot arm.

**ACTUATORS** — The drive mechanisms that position the manipulator to a predetermined point. These actuators can be pneumatic or hydraulic cylinders, pneumatic or hydraulic rotary motors, or electric motors. Specific actuators and methods of controlling them will be covered in detail in Unit Two.

**GRIPPER** — A hand-like device sometimes referred to as the “end-effector”. It holds the tool doing the work or handles the material being worked on. Most grippers are simple open/close devices that are actuated hydraulically, pneumatically, mechanically, or with electric motors. The variety of grippers that can be adapted for robot use is virtually unlimited.

**CONTROLLER** — The brains and nervous system of the robot. It initiates and terminates the movements of the robot, stores position and sequence data, and interfaces the robot to the “outside world”. A controller can be any programmable device from a simple mechanical rotary drum switch to a complex electronic computer.

**POWER SUPPLY** — Provides and regulates the energy for the robot’s actuators. This energy can be pneumatic, hydraulic, electrical, electro-hydraulic, and in some cases even, mechanical.

**CYLINDRICAL COORDINATE ROBOT** — Essentially, a horizontal manipulator mounted on a vertical column. The vertical column, in turn, is mounted on a rotating base. This is shown in Figure 1-6. The manipulator is free to extend and retract, and is also free to move up and down on the vertical column; both of the members are permitted to rotate on the base. Thus, the work envelope (working area) of the robot forms a portion of a cylinder.

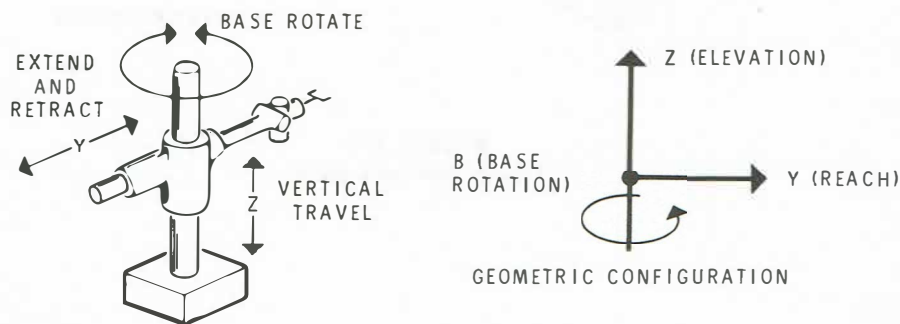


Figure 1-6  
Cylindrical coordinate robot.

**SPHERICAL COORDINATE ROBOT** — This type of robot, as shown in Figure 1-7, can be compared to the turret of a tank. The manipulator extends and retracts, pivots in the vertical plane, and rotates in the horizontal plane about the base. The work envelope forms a small portion of a sphere.

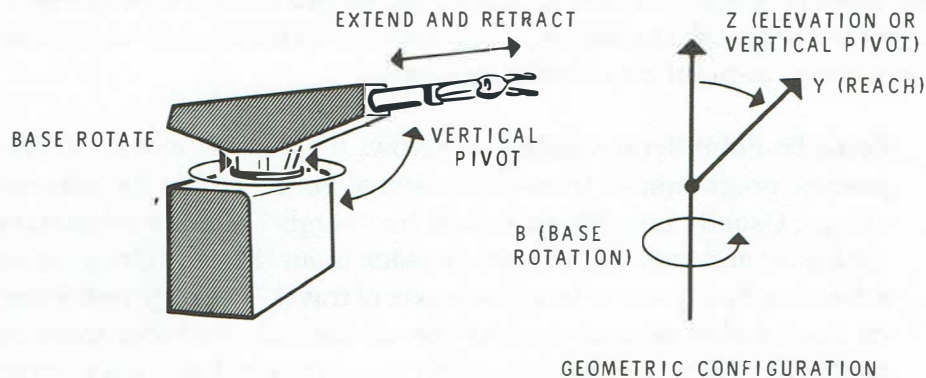


Figure 1-7  
Spherical coordinate robot.

**JOINTED-SPHERICAL COORDINATE ROBOT** — A base plus an upper arm and forearm which move in the vertical plane through the base. A jointed-spherical coordinate robot is shown in Figure 1-8. The manipulator has an “elbow” joint located between the forearm and the upper arm, plus a “shoulder” joint located between the upper arm and the base. Rotary motion in the horizontal plane can be provided either at the shoulder joint or at the bottom of the base. The work envelope forms a major portion of a sphere.

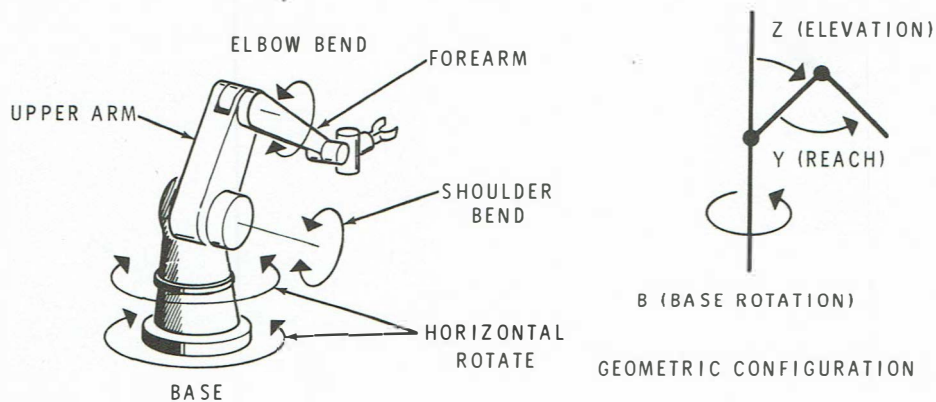


Figure 1-8  
Jointed-spherical coordinate robot.

**NON-SERVO ROBOTS** — Robots that are restricted to three or four degrees of freedom. Mechanical stops (end stops) are used on each axis to limit the amount of travel. Thus, there are usually only two positions for each axis to assume: up/down, left/right, extend/retract.

**SERVO-CONTROLLED ROBOTS** — Robots whose axes of motion are controlled to move and to stop anywhere within their limits of travel, rather than only at the extremes. There are two classes of servo-controlled robots: point-to-point and continuous path.

**Point-To-Point Servo Control** — Allows the robot to move between precise, programmed, three-dimensional points within its work envelope. Usually only the end stops are “taught” and the robot takes the fastest and most direct route to reach them. The end stops can be set or taught anywhere in a given axis of travel. The only restriction on the number of end stops is the amount of controller memory available. Point-to-point operation is always a little jerky, even when two axes are controlled simultaneously. For this reason, point-to-point control is used where only the final position is important and the path and velocity between programmed points are not major considerations.

**Continuous Path Servo Control** — Allows the robot to smoothly follow a particular path rather than accelerate sharply between end stops. The trajectory of the continuous path robot is usually a curved path with little or no stopping at preprogrammed end stops. For this reason, continuous path robots are ideally suited for spray painting applications. One drawback to the continuous path servo-controlled robot is the large amount of controller memory required. Greater memory is required to store all axes positions needed for the robot to smoothly follow the desired path.



## Programmed Review

9.	The number of _____ a robot has determines the complexity of a task the robot can perform.
10.	(axes/degrees of freedom) The _____ is a system of mechanical linkages and joints that performs the robot's work.
11.	(manipulator) The _____ drive the robot to a pre-determined point.
12.	(actuators) The _____ is used to interface the robot with the outside world.
13.	(controller) A _____ robot has a work envelope that is a portion of a cylinder.
14.	(cylindrical coordinate) A jointed-spherical coordinate robot has an _____ joint plus a _____ joint to permit greater manipulator flexibility.
15.	(elbow, shoulder) On non-servo robots, _____ are used to limit the amount of travel in each axis.
16.	(mechanical stops/end stops) The two classes of servo-controlled robots are _____ and _____.
17.	(point-to-point, continuous path) In point-to-point control of a robot, the _____ can be set or taught anywhere in a given axis.
18.	(end stops) When a robot is being controlled by continuous path servo-control, a large amount of controller _____ is required.
	(memory)

## LOW-TECHNOLOGY ROBOTS

Figure 1-9 shows the three categories of industrial robots: low-technology, medium-technology, and high-technology. Each robot category has its own unique characteristics, such as: number of axes of motion, type of controller, method of controller programming, type of manipulator motion, and kind of actuation used. Under the general classification of industrial robots, these categories can be based on the type of work the robot is assigned to do. Figure 1-10 shows some industrial tasks assigned to robots and the technology level required to complete the task. This section is devoted to the low-technology robot, while later sections discuss the medium and high-technology robots.

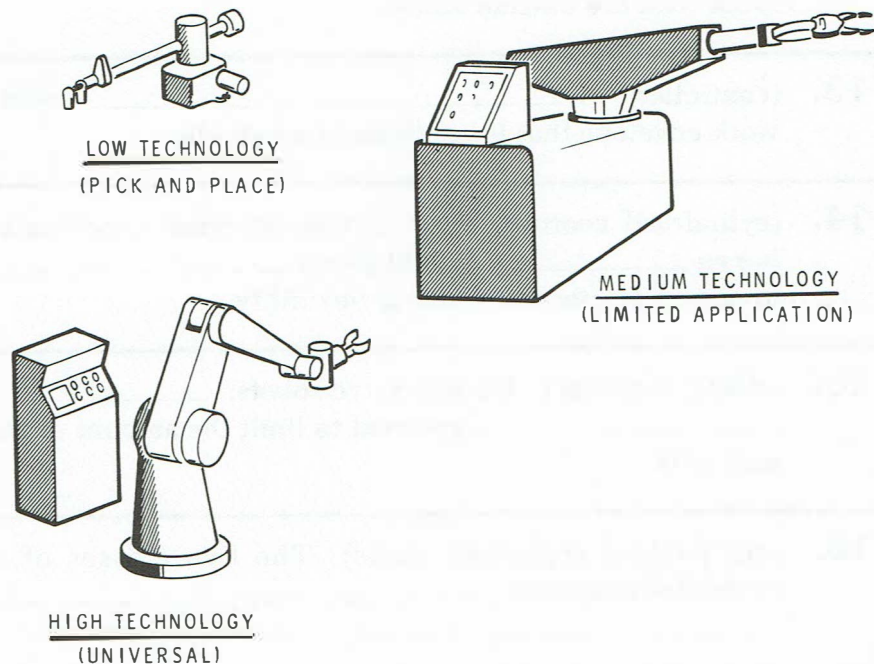


Figure 1-9  
Types of industrial robots.

TASK ASSIGNED	TECHNOLOGY LEVEL		
	Low	Medium	High
Material Handling	X	X	X
Press Operations	X	X	X
Injection Molding	X	X	X
Machine Load & Unload	X	X	X
Assembly (simple)	X	X	X
Die Casting		X	X
Spray Painting			X
Arc Welding			X
Spot Welding			X
Forging			X
Palletizing			X
Inspection (Body & Panels)			X
Complex Parts Assembly			X

Figure 1-10  
Robot classification by task.

Low-technology robots, such as the one shown in Figure 1-11, are also called “pick-and-place” and “limited sequence” devices. While this type of robot is the simplest version, it accounts for nearly 35% of all U.S. industrial robots in operation today. Virtually all low-technology robots fall in the “non-servo” robot class. They differ from hard automation in that you can alter their pattern of movement somewhat by moving the end stops. Thus, the amount, sequence, and to some extent speed of travel, in a given axis, can be controlled.

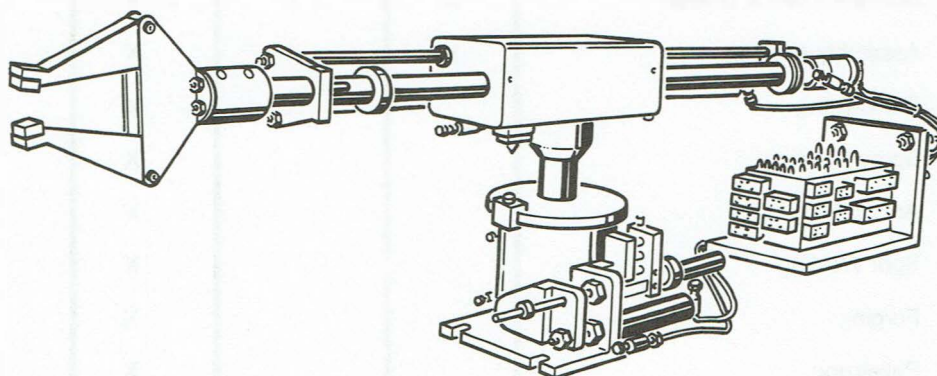


Figure 1-11  
Low-technology robot, Auto-Place Series 50.

## Characteristics

There are several characteristics which distinguish the low-technology industrial robot from the more sophisticated industrial robots. Let's take a few minutes and look at some of these characteristics.

### AXES (DEGREES OF FREEDOM)

Most low-technology robots have between two and four non-servo degrees of freedom. Mechanical end stops are used on each axis to limit the amount of travel in a given direction. There are usually only two positions for each axis to assume: up/down, extend/retract, left/right.



The shaded area of Figure 1-12 shows the number of axes of motion of three typical pick-and-place robots. As you can see, the number of axes of motion is limited for this type of robot: two for the Seiko Model 400L, three for the Auto-Place Series 50, and four for the Auto-Mate Standard Model. The maximum distance of travel along each axis is also restricted in comparison to more sophisticated robots.

	SEIKO Model 400L	AUTO-PLACE Series 50	AUTO-MATE Standard Model
VERTICAL (up/down)	20 - 100 mm (.79" - 3.94")	0 - 127 mm (0 - 5")	0 - 178 mm (0 - 7")
HORIZONTAL (extend/ retract)	0 - 700 mm (0 - 27.56")	0 - 457 mm (0 - 18")	0 - 610 mm (0 - 24")
SWING (left/right)	NONE	0 - 200 degrees	90 or 120 degrees
ELBOW(Shoulder)			10 degrees
PAYLOAD	3 kg (6.61 lbs)	13.6 kg (30 lbs)	4.5 kg (10 lbs)
SPEED *	.6 sec vertical axis .7 sec horizontal axis	1000 mm/sec (40"/sec)	.8 sec in all axes of travel
ACCURACY	±.050 mm (±.002")	±.025 mm (±.001")	NOT AVAILABLE
ACTUATION	Pneumatic	Pneumatic Hydraulic	Air/oil
CONTROLLER	Mechanical or Electronic	Air-logic or Solid State Programmable	Air-logic or Solid State Programmable

\*All speed characteristics are for maximum distance of travel.

**Figure 1-12**  
Axes of motion  
(shaded)

The Seiko Model 400L, as shown in Figure 1-13, has a vertical axis of travel adjustable between 20-100 mm. The horizontal axis of travel is adjustable between 0 (fully retracted) and 700 mm (fully extended). This robot does not have the capability of swinging either left or right, but the gripper is capable of rotating 180°, and 50 mm of gripper movement along a lateral axis is also possible.

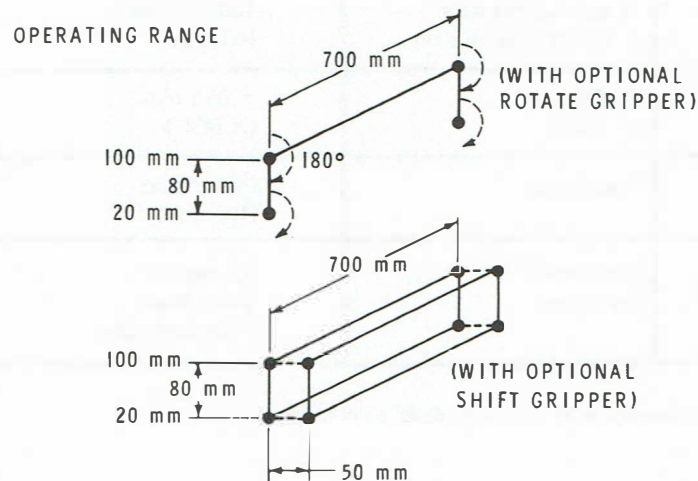
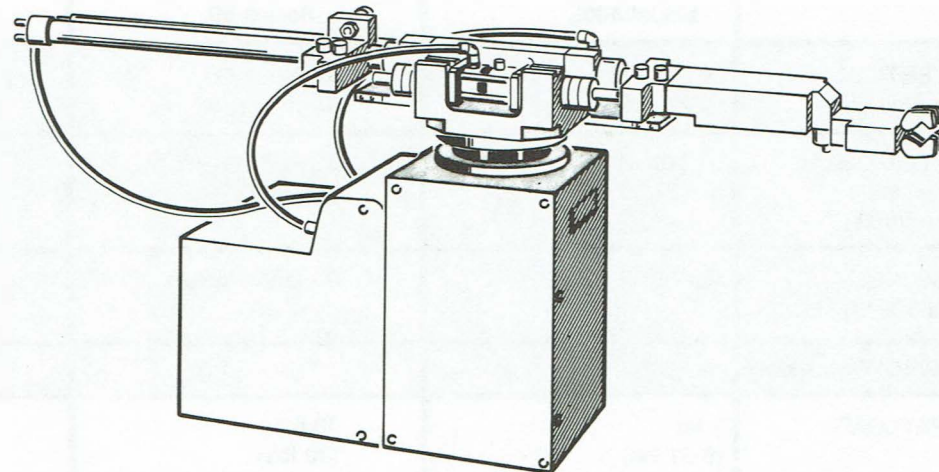


Figure 1-13  
Seiko Model 400L with optional gripper  
shift and rotate.

The Auto-Mate robot shown in Figure 1-14, has three standard axes of motion common to limited-sequence devices: vertical, horizontal, and swing. In addition, a fourth degree of freedom, shoulder/elbow, has also been added to increase the robot's capabilities. The vertical limit of travel is adjustable between 0-178 mm, but a total vertical motion of 356 mm is possible using the  $10^\circ$  of freedom available at the shoulder.

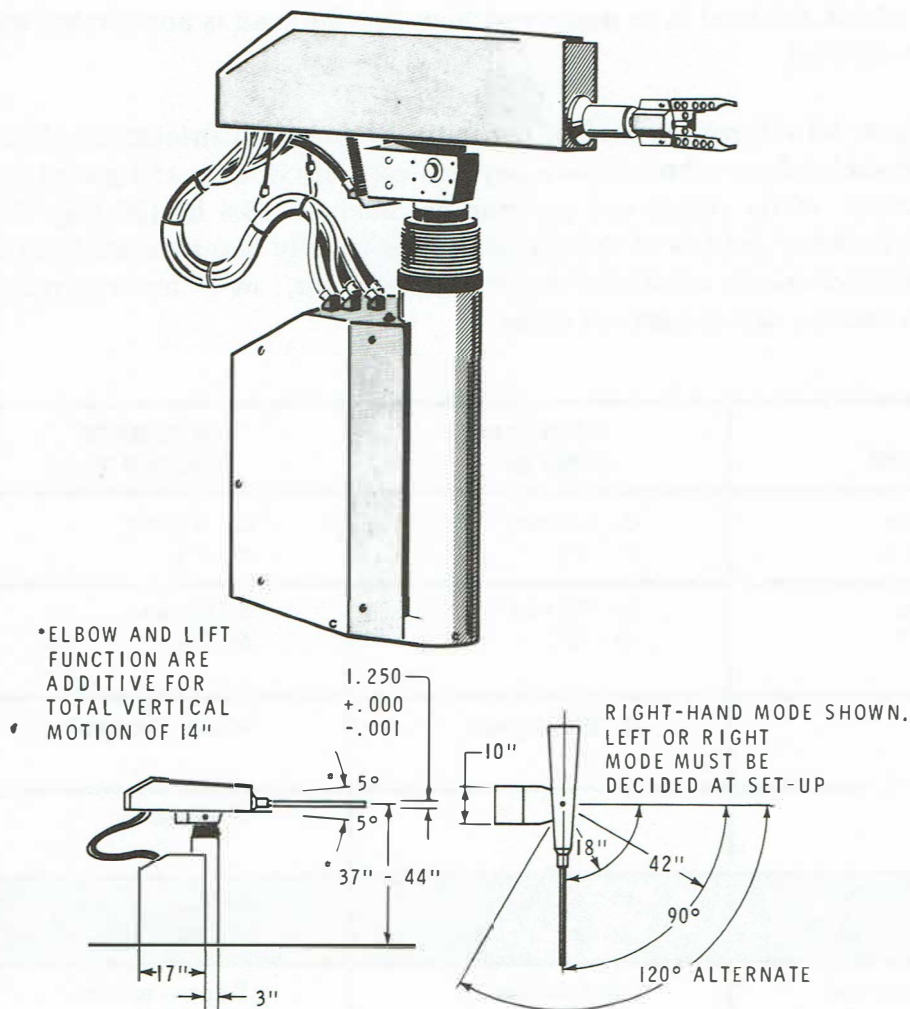


Figure 1-14  
Auto-Mate Robot with four degrees of freedom.

The robot manipulator is adjustable between 457 mm (fully retracted) and 1067 mm (fully extended). This permits a total of 610 mm adjustable motion along the horizontal axis.

The amount of swing is either  $90^\circ$  or  $120^\circ$  and is not adjustable. Whether the robot moves left or right must be decided during the initial set-up.

## PAYLOAD

Payload is the maximum load capacity a manipulator can position. This capacity is measured at the gripper. Many robot manufacturers' specifications give the load capacity under static (stationary) conditions. This is not necessarily the load the arm can handle at maximum reach and speed. Maximum load depends not only on the strength of the arm but the speed at which the load is to move and how fast the load is accelerated and decelerated.

Figure 1-15 shows the payload capacities of three industrial robots. Some low-technology robots have a payload capacity of only 150 grams (5.3 ounces) while others can position as much as 13.6 kg (30 lbs). The payload capabilities of simple robots are usually less than medium or high-technology robots; but as you will see later, low-technology robots are usually much more accurate.

	SEIKO Model 400L	AUTO-PLACE Series 50	AUTO-MATE Standard Model
VERTICAL (up/down)	20 - 100 mm (.79" - 3.94")	0 - 127 mm (0 - 5")	0 - 178 mm (0 - 7")
HORIZONTAL (extend/ retract)	0 - 700 mm (0 - 27.56")	0 - 457 mm (0 - 18")	0 - 610 mm (0 - 24")
SWING (left/right)	NONE	0 - 200 degrees	90 or 120 degrees
ELBOW (Shoulder)			10 degrees
<b>PAYLOAD</b>	<b>3 kg (6.61 lbs)</b>	<b>13.6 kg (30 lbs)</b>	<b>4.5 kg (10 lbs)</b>
SPEED	.6 sec vertical axis .7 sec horizontal axis	1000 mm/sec (40"/sec)	.8 sec in all axes of travel
ACCURACY	±.050 mm (±.002")	±.025 mm (±.001")	
ACTUATION	Pneumatic	Pneumatic or Hydraulic	Air/oil
CONTROLLER	Mechanical or Electronic	Air-logic or Solid State Programmable	Air-logic or Solid State Programmable

Figure 1-15  
Payload capacities  
(shaded).



## SPEED

The speed at which the manipulator travels determines the cycle time of a robot, the amount of time required to perform one complete sequence of operations. The amount of payload and length of the manipulator are major factors in determining the speed.

Figure 1-16 shows the speed characteristics for maximum distance of travel of the three examples of low-technology robots. These robots have a faster cycle time than most of their more sophisticated counterparts. Therefore, what they lack in payload capability they make up for in speed. One important item to remember: in relation to speed, the robot must be able to meet or exceed the cycle time of the machine it serves.

	SEIKO Model 400L	AUTO-PLACE Series 50	AUTO-MATE Standard Model
VERTICAL (up/down)	20 - 100 mm (.79" - 3.94")	0 - 127 mm (0 - 5")	0 - 178 mm (0 - 7")
HORIZONTAL (extend/ retract)	0 - 700 mm (0 - 27.56")	0 - 457 mm (0 - 18")	0 - 610 mm (0 - 24")
SWING (left/right)	NONE	0 - 200 degrees	90 or 120 degrees
ELBOW (Shoulder)			10 degrees
PAYLOAD	3 kg (6.61 lbs)	13.6 kg (30 lbs.)	4.5 kg (10 lbs)
SPEED	.6 sec vertical axis .7 sec horizontal axis	1000 mm/sec (40"/sec)	.8 sec in all axes of travel
ACCURACY	±.050 mm (±.002")	±.025 mm (±.001")	
ACTUATION	Pneumatic	Pneumatic or Hydraulic	Air/oil
CONTROLLER	Mechanical or Electronic	Air-logic or Solid State Programmable	Air-logic or Solid State Programmable

Figure 1-16  
Speed characteristics  
(shaded).

## ACCURACY

Accuracy is defined as the placement repeatability of the robot. Simply stated, accuracy is determined by how close the robot can position its manipulator to a given point. Repeatability is the robot's ability to return its manipulator to the same given point when repeatedly cycled. Accuracy figures apply at any performance level within each robot's payload, cycle frequency, and work envelope capabilities.

Small pick-and-place robots, like those listed in Figure 1-17, are capable of a much higher degree of accuracy than larger robots. For example, placement and repeatability from  $\pm .025$  mm to  $\pm .2$  mm is common, with some, low-technology robots capable of accuracies of  $\pm .01$  mm.

	SEIKO Model 400L	AUTO-PLACE Series 50	AUTO-MATE Standard Model
VERTICAL (up/down)	20 - 100 mm (.79" - 3.94")	0 - 127 mm (0 - 5")	0 - 178 mm (0 - 7")
HORIZONTAL (extend/ retract)	0 - 700 mm (0 - 27.56")	0 - 457 mm (0 - 18")	0 - 610 mm (0 - 24")
SWING (left/right)	NONE	0 - 200 degrees	90 or 120 degrees
ELBOW (Shoulder)			10 degrees
PAYLOAD	3 kg (6.61 lbs)	13.6 kg (30 lbs)	4.5 kg (10 lbs)
SPEED	.6 sec vertical axis .7 sec horizontal axis	1000 mm/sec (40"/sec)	.8 sec in all axes of travel
ACCURACY	$\pm .050$ mm ( $\pm .002$ ")	$\pm .025$ mm ( $\pm .001$ ")	
ACTUATION	Pneumatic	Pneumatic or Hydraulic	Air/oil
CONTROLLER	Mechanical or Electronic	Air-logic or Solid State Programmable	Air-logic or Solid State Programmable

Figure 1-17  
Accuracy of pick-and-place robots  
(shaded).

## ACTUATION

As we stated previously, robots are actuated, or positioned, either hydraulically, pneumatically, or by electric motors. Some robots are positioned using a combination of these methods. Very basic robots may be actuated mechanically through bell cranks and cams. Mechanically actuated robots will not be discussed further, since robots of this type are often classified as automation.

In general, pneumatics are best suited for fast movements and light payloads, two characteristics of limited-sequence robots. Hydraulic actuation offers the ability to position and hold heavy loads for sustained periods without slip. On the other hand, electric drives are considered to be the simplest to control.

Most limited-sequence robots, like those listed in Figure 1-18, are pneumatically actuated. One major advantage of pneumatic actuation over hydraulic actuation is cost. Pneumatic operated systems can be supplied by factory air, whereas hydraulic operated systems usually require a separate hydraulic power source for each individual unit.

	SEIKO Model 400L	AUTO-PLACE Series 50	AUTO-MATE Standard Model
VERTICAL (up/down)	20 - 100 mm (.79" - 3.94")	0 - 127 mm (0 - 5")	0 - 178 mm (0 - 7")
HORIZONTAL (extend/ retract)	0 - 700 mm (0 - 27.56")	0 - 457 mm (0 - 18")	0 - 610 mm (0 - 24")
SWING (left/right)	NONE	0 - 200 degrees	90 or 120 degrees
ELBOW (Shoulder)			10 degrees
PAYLOAD	3 kg (6.61 lbs)	13.6 kg (30 lbs)	4.5 kg (10 lbs)
SPEED	.6 sec vertical axis .7 sec horizontal axis	1000 mm/sec (40"/sec)	.8 sec in all axes of travel
ACCURACY	±.050 mm (±.002")	±.025 mm (±.001")	
ACTUATION	Pneumatic	Pneumatic or Hydraulic	Air/oil
CONTROLLER	Mechanical or Electronic	Air-logic or Solid State Programmable	Air-logic or Solid State Programmable

Figure 1-18  
Methods of actuation  
(shaded).



An interesting “hybrid” form of actuation is used in the Auto-Mate. Quite simply, it uses factory air to actuate oil filled cylinders which, in turn, produce manipulator motion. Hydraulic, pneumatic, and electric drive systems will be covered in greater detail later in this course.

## CONTROLLERS

The key distinction between robots and hard automation is in their learning capability. Low-technology robots are hard-automation devices that you can control by setting stops and adjusting cams or sequencing valves, but devices that cannot learn. Limited-sequence robots are usually very limited in the amount of information that can be stored in their memory. As a rule, only sequence and time are used in their programs, although some branching and subroutines are possible with today’s low cost, increasingly powerful mini-computer and microprocessor controllers.

Figure 1-19 lists some of the basic controllers being used with low-technology robots. Because they are limited-sequence devices dedicated to performing specific tasks, large amounts of positioning data and memory are not required; hence, the simple controllers.

The table is extremely faded and illegible. It appears to have approximately 4 columns and 10 rows. The text within the cells is not readable.

	<b>SEIKO Model 400L</b>	<b>AUTO-PLACE Series 50</b>	<b>AUTO-MATE Standard Model</b>
<b>VERTICAL</b> (up/down)	20 - 100 mm (.79" - 3.94")	0 - 127 mm (0 - 5")	0 - 178 mm (0 - 7")
<b>HORIZONTAL</b> (extend/ retract)	0 - 700 mm (0 - 27.56")	0 - 457 mm (0 - 18")	0 - 610 mm (0 - 24")
<b>SWING</b> (left/right)	NONE	0 - 200 degrees	90 or 120 degrees
<b>ELBOW</b> (Shoulder)			10 degrees
<b>PAYLOAD</b>	3 kg (6.61 lbs)	13.6 kg (30 lbs)	4.5 kg (10 lbs)
<b>SPEED</b>	.6 sec vertical axis .7 sec horizontal axis	1000 mm/sec (40"/sec)	.8 sec in all axes of travel
<b>ACCURACY</b>	±.050 mm (±.002")	±.025 mm (±.001")	
<b>ACTUATION</b>	Pneumatic	Pneumatic or Hydraulic	Air/oil
<b>CONTROLLER</b>	<b>Mechanical or Electronic</b>	<b>Air-logic or Solid State Programmable</b>	<b>Air-logic or Solid State Programmable</b>

Figure 1-19  
Low-technology robot control methods  
(shaded).

Controllers for the Seiko robots may be either mechanical or electronic. In their simplest form, a controller need only execute a fixed-sequence program. A controller should be chosen depending on the demands of the task that the robot must perform. The mechanical controller shown in Figure 1-20A is sufficient for simple stand-alone applications. A typical application would be a single robot, with a short program sequence and with few peripheral control circuits, feeding a press. For more complex integrated systems that require long program sequences, multiple closed-loop control circuits, and programming flexibility, an electronic programmable controller is more than sufficient.

An Auto-Place robot is usually controlled by an air-logic control system, as shown in Figure 1-20B. This logic control system dictates sequential robot motions that are activated by small valves arranged in a circuit on a sequencing module. When one motion is completed, the next is initiated.

Since the control proceeds sequentially through a given series of motions to complete a full cycle, the series of actions can be interrupted at any point by sensors, limit switches, or other external inputs.

While air-logic is the primary means of controlling Auto-Place robots, solid state or hard wired controllers are also available.

The Auto-Mate, another robot in the limited sequence category, uses either air-logic or solid-state controllers. With a solid-state controller, as shown in Figure 1-20C, the desired program steps are first calculated on a special chart. Once the program steps have been calculated, a series of buttons are pressed to enter them into the controller. This programmable controller has the capability of storing 256 separate program steps in memory.

As you can see, the controller is the robot's most essential component — its brain! Without a brain, a robot would be just another machine. For this reason, controllers and controller programming will be discussed in great detail throughout this course.

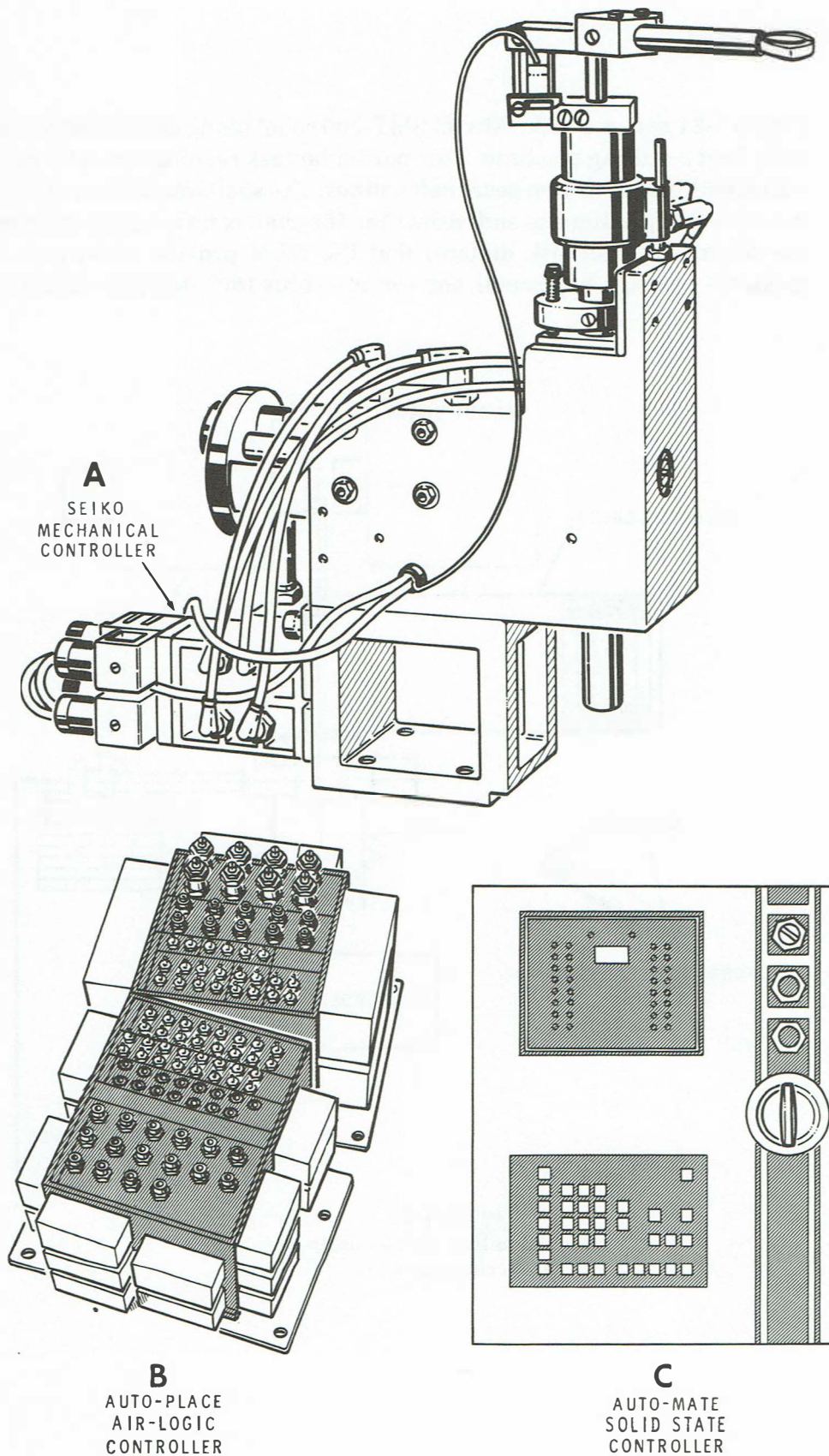


Figure 1-20  
Basic controllers for low-technology robots.



## Operation

Figure 1-21 shows a Seiko Model SMT-700 robot being used to automatically feed a milling machine. This particular task requires the milling of automobile transmission gear shaftsplines. The shafts are fed to a milling machine for splining one end. After that, the shaft is reversed for splining the other end. The task dictates that the robot provide three axes of motion — vertical, horizontal, and swing — plus 180° of gripper rotation.

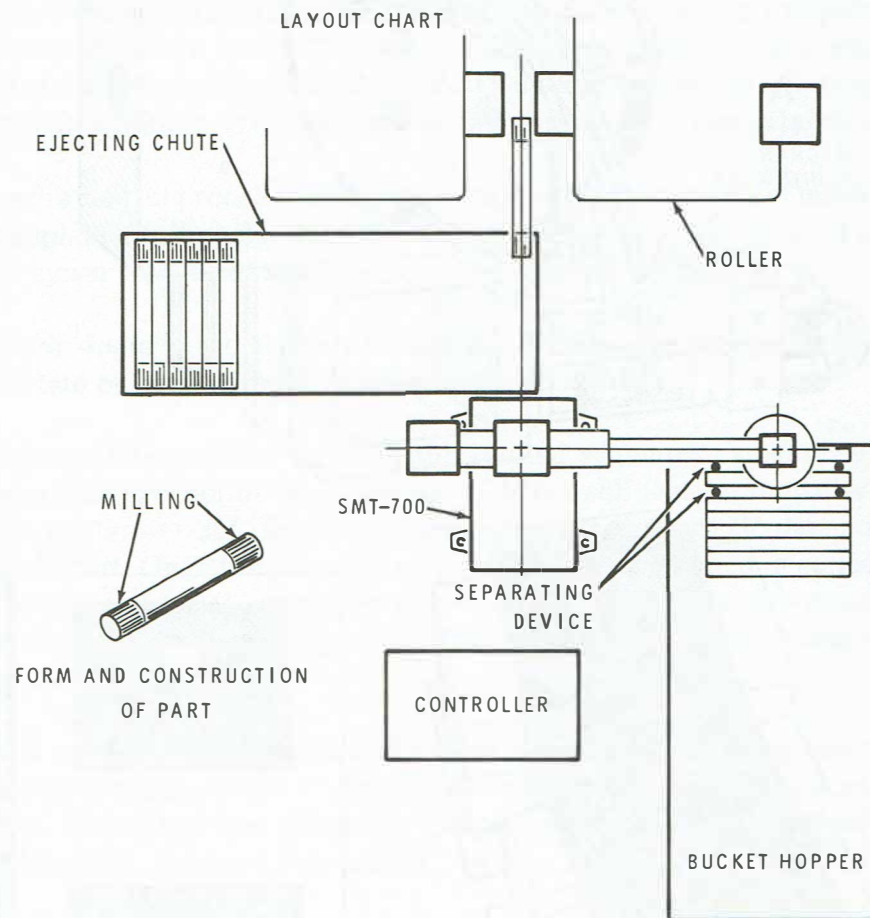


Figure 1-21  
Industrial milling process using a  
low-technology robot.

The gear shafts are loaded into a bucket hopper after preprocessing and separated automatically one by one. The robot is now free to operate as follows:

<u>STEP NUMBER</u>	<u>OPERATION</u>
1.	Manipulator lowers.
2.	Gripper grasps shaft.
3.	Manipulator rises and retracts.
4.	Robot rotates 90° counterclockwise.
5.	Manipulator extends.
6.	Manipulator lowers.
7.	Gripper releases shaft.
8.	Manipulator rises (milling takes place).
9.	Manipulator lowers.
10.	Gripper grasps shaft.
11.	Manipulator rises and retracts.
12.	Gripper rotates (shaft turns 180°).
13.	Manipulator extends.
14.	Manipulator lowers.
15.	Gripper releases shaft.
16.	Manipulator rises (milling takes place).
17.	Manipulator lowers.
18.	Gripper grasps shaft.
19.	Manipulator rises and retracts.
20.	Gripper releases shaft (onto ejecting chute).
21.	Robot rotates 90° clockwise.
22.	Manipulator extends.
	Cycle is repeated.

As you can see, this operation requires only 22 separate steps to complete. We must emphasize that, because low-technology robots are very limited in the number of moves they can perform, they are very dependent on support equipment such as bowl feeders and part presenters.

## Programmed Review

19. Virtually all low-technology robots fall into the \_\_\_\_\_ robot class.

20. (non-servo) In low-technology robots, there are usually only \_\_\_\_\_ positions for each axis to assume.

21. (two) The maximum distance of travel along each axis of a low-technology robot is usually \_\_\_\_\_ than that of the more sophisticated robot. (less/greater)

22. (less) The payload a robot can effectively carry is measured at the \_\_\_\_\_.

23. (gripper) Most low-technology robots have a \_\_\_\_\_ cycle time than the more sophisticated robots. (faster/slower)

24. (faster) When we speak of a robot's accuracy, we refer to \_\_\_\_\_ as the robot's ability to return to a given point when recycled.

25. (repeatability) Most pick-and-place robots are \_\_\_\_\_ actuated due to their light payloads.

26. (pneumatically) Hydraulic actuation is generally \_\_\_\_\_ expensive than pneumatic actuation. (more/less)

27. (more) Because the low-technology robot has a very limited amount of memory available, usually only \_\_\_\_\_ and \_\_\_\_\_ are used in its program.

28. (sequence, time) The \_\_\_\_\_ control system uses a series of small valves positioned on a sequencing module.

29. (air-logic) The \_\_\_\_\_ is the robot's most essential component.

(controller)



## MEDIUM AND HIGH TECHNOLOGY ROBOTS

You will now study the more sophisticated medium and high technology robots. These two categories of robots will not be discussed in great detail as was the low-technology robot. You have now acquired most of the basic principles associated with robotics and these principles apply to all robots. The more sophisticated robots are presented for comparison purposes. With this in mind, we will now study the more sophisticated members of the robot family — the medium and high technology robot.

### The Medium-Technology Robot

The medium-technology robot of today was the sophisticated robot of just a few years past. Just as the pick-and-place robot has its place in industry, so does the medium-technology robot. These robots are used in machine load/unload or material transfer jobs where point-to-point operation is desired.

Figure 1-22, on Page 1-34, shows the characteristics of two medium-technology robots. At first glance, it may appear that the characteristics are the same as those of the low-technology robot. That is not the case. To aid in better understanding these differences, Figure 1-23, shown on Page 1-35, compares the low and medium-technology robot. refer back occasionally to Figure 1-23 when you are comparing the two categories of robots.

	UNIMATE 1000	PRAB 4200HD
HORIZONTAL (extend/retract)	1.8 - 2.24 m (46-1/2" - 88-1/4")	1 m - 2.08 m (39-1/2" - 82")
SWING (left/right)	208 degrees	250 degrees
ELEVATION (shoulder)	30 deg above vert trav 26 deg below vert trav	10 deg above vert trav 10 deg below vert trav
WRIST BEND	90 deg within 180 deg	90 deg within 180 deg
WRIST YAW	90 deg	
WRIST ROTATE		180 degrees
PAYLOAD	22.7 kg (50 lbs)	56.7 kg (125 lbs)
SPEED	914 mm/sec (36"/sec)	1.02 m/sec (40"/sec)
ACCURACY	± 1.27 mm ± (.05")	± .2 mm ± (.008")
ACTUATION	Hydraulic (pneumatic gripper)	Electro-hydraulic (mechanical gripper)
CONTROLLER	Solid State	Rotating Drum

Figure 1-22  
Characteristics of medium-technology robots.

## LOW TECHNOLOGY ROBOT

	SEIKO Model 400L	AUTO-PLACE Series 50	AUTO-MATE Standard Model
VERTICAL (up/down)	20 - 100 mm (.79" - 3.94")	0 - 127 mm (0 - 5")	0 - 178 mm (0 - 7")
HORIZONTAL (extend/ retract)	0 - 700 mm (0 - 27.56")	0 - 457 mm (0 - 18")	0 - 610 mm (0 - 24")
SWING (left/right)	NONE	0 - 200 degrees	90 or 120 degrees
ELBOW (Shoulder)			10 degrees
PAYLOAD	3 kg (6.61 lbs)	13.6 kg (30 lbs)	4.5 kg (10 lbs)
SPEED	.6 sec vertical axis .7 sec horizontal axis	1000 mm/sec (40"/sec)	.8 sec in all axes of travel
ACCURACY	±.050 mm (±.002")	±.025 mm (±.001")	
ACTUATION	Pneumatic	Pneumatic or Hydraulic	Air/oil
CONTROLLER	Mechanical or Electronic	Air-logic or Solid State Programmable	Air-logic or Solid State Programmable

## MEDIUM-TECHNOLOGY ROBOT

	UNIMATE 1000	PRAB 4200HD
HORIZONTAL (extend/retract)	1.8 - 2.24 m (46-1/2" - 88-1/4")	1 m - 2.08 m (39-1/2" - 82")
SWING (left/right)	208 degrees	250 degrees
ELEVATION (shoulder)	30 deg above vert trav 26 deg below vert trav	10 deg above vert trav 10 deg below vert trav
WRIST BEND	90 deg within 180 deg	90 deg within 180 deg
WRIST YAW	90 deg	
WRIST ROTATE		180 degrees
PAYLOAD	22.7 kg (50 lbs)	56.7 kg (125 lbs)
SPEED	914 mm/sec (36"/sec)	1.02 m/sec (40"/sec)
ACCURACY	1.27 mm (.05")	.2 mm (.008")
ACTUATION	Hydraulic (pneumatic gripper)	Electro-hydraulic (mechanical gripper)
CONTROLLER	Solid State	Rotating Drum

Figure 1-23  
Comparison of low and  
medium-technology robots.

## AXES

As you can see, medium-technology robots not only have more axes of motion, they are also able to travel a greater distance in each axis than the limited-sequence robot. Whereas low-technology robots have between two and four axes of freedom, medium-technology robots can have up to six. The vertical axis has been replaced with an elevation (shoulder) axis, and both the horizontal and swing axes have a greater amount of freedom of travel. This greater freedom allows the mid-technology robot to have the capability of serving more than one machine at a time.

The medium-technology robot also has a greater degree of freedom at the end of the manipulator due to wrist bend, wrist yaw, and wrist rotate. Some medium-technology robots use end-stops to limit the amount of travel in each axis, while others have the capability of making several stops along a given axis.

## PAYLOAD

As you can see from Figure 1-23, the payload capability of the medium-technology robot is much greater than that of the limited-sequence robot. This greater payload capacity greatly increases the usefulness of the robot. For example, the robot can now be used to extract a heavy object from one machine and load it into another, load objects into a blast furnace, and generally work at tasks that are taxing to a human. The maximum payload a medium-technology robot can be expected to handle effectively is approximately 68 kg, or 150 lbs.

## SPEED

Again referring to Figure 1-23, you can see that the speed at which a medium-technology robot can work is nearly the same as that of the low-technology robot. One important factor to remember is that while the speeds are relatively the same for both robots, the cycle time for the medium-technology robot may be much slower. This is due to the nature of the tasks performed by the more sophisticated robot. Most tasks performed by the simpler robots required handling small objects for relatively short machine work cycles, such as the Seiko robot presenting work to a milling machine. The more advanced robot, due to the nature of its task, presents a type of work to a machine that requires a much longer machine work cycle. Thus, the cycle time of the medium-technology robot would be much greater.

## ACCURACY

The medium-technology robot, as described in Figure 1-23, is not as accurate as the limited sequence robot; but it is more than sufficient to accomplish the type of task it was designed for. Two factors that you must take into consideration when you compare accuracy tolerances of various robots are speed and payload capabilities, both of which greatly effect the accuracy of a robot.

## ACTUATION

Medium-technology robots are usually actuated by hydraulic or electro-hydraulic means; refer to Figure 1-22. As you will recall, the simpler robots were positioned by pneumatic actuation. This was due mostly to the light payloads they had to position, and also to keep their cost at a minimum. But once a certain payload limit has been reached, pneumatics are no longer feasible or cost effective. For this reason, hydraulics or electro-hydraulics are the preferred methods of actuation for medium-technology robots. Hydraulics and hydraulic actuation will be discussed in greater detail later in this course.

## CONTROLLER

Most of the characteristics of the medium-technology robot that we have talked about so far have been similar to those of the low-technology robot. The most important difference between the two categories of robots is in the method of controlling them. The limited-sequence robot did not have the ability to learn whereas the medium-technology robot has that capability. In order to change the sequence of motions of a simple robot, it was necessary to change mechanical stops. To change the sequence of motion for a medium-technology robot, or to install a complete new sequence, is a relatively simple operation.

The two medium-technology industrial robots of Figure 1-22 are controlled by completely different methods. The Unimate 1000 uses a solid-state controller capable of storing 32 separate steps in its memory. If you want to change the controller sequence or use a completely different sequence, it's simply a matter of reteaching the robot or substituting a previously stored program.



The “rotating drum”, shown in Figure 1-24, is a unique controller used by Prab Conveyors Inc. to guide their medium-technology robots through a desired sequential operation. Cams mounted on the drum make contact with microswitches during drum rotation. The switches send signals which open hydraulic valves that drive the robot through its various motions: rotation, elevation, extend/retract, and various wrist movements.

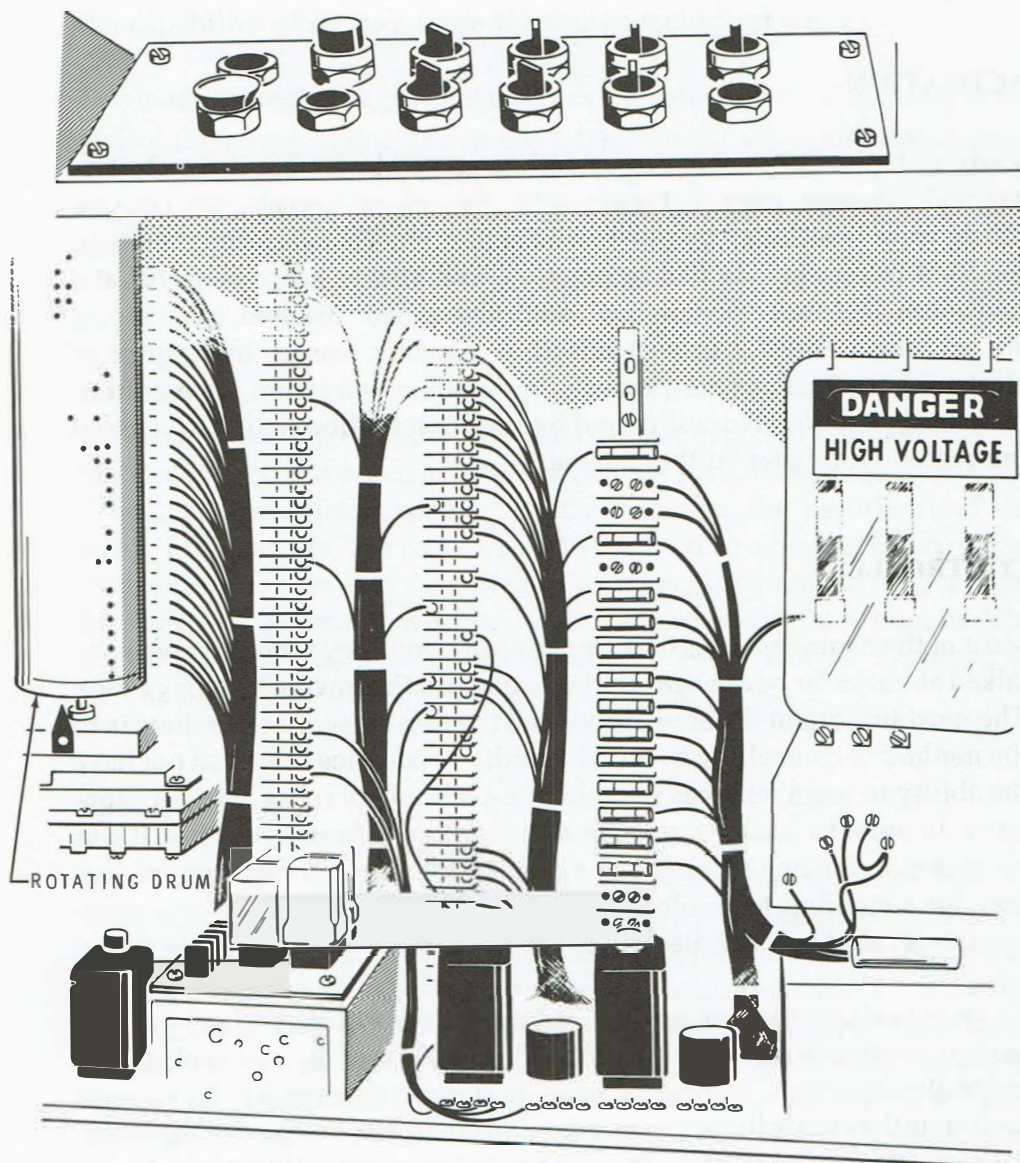


Figure 1-24  
PRAB, Programmable Rotating Drum.

The rotating drum gives the robot the ability to perform up to 60 separate functions in a single cycle. These rotating drums are easily inserted and removed from the control console. Extra drums can be kept on hand so that, once a task is programmed, the drum is seldom changed. If the robot is assigned another task, the drum can be removed and stored until the same task is required again.

While the drum governs the sequence of robot movements, the speed at which the movements are performed is controlled from the control console. The operator simply sets the desired speed by pushing a button.

We should also mention that Prab includes a microprocessor-based controller in their list of optional equipment, thereby eliminating the need for the rotating drum.



## High-Technology Robots

High-technology industrial robots are the elite members of the robot family — at the leading edge of technology. Robots of this type have highly flexible manipulators and use advanced programmable controllers. These robots can perform numerous industrial tasks such as spray-painting, complicated welding procedures, and assembly operations; just to name a few. The characteristics of these high-technology robots will not be presented in the same manner as the characteristics for the low and medium-technology robots. Instead, we will examine one family of high-technology robots in detail: the PUMA® (Programmable Universal Manipulator for Assembling), manufactured by Unimation Inc., of Danbury, CT. Many of the terms associated with the PUMA robot have already been explained, any new terms will be explained as we proceed.

### PUMA FAMILY

The PUMA family actually consists of three robots: the 250, 500, and 600 series. These are sophisticated, programmable, microprocessor-controlled robots designed for close tolerance assembly operations and small parts handling. The PUMA robots' motions are similar to those of the human body and can be described in human terms: waist, shoulder, elbow, wrist, hand rotation and waist bend. They are designed to work with humans, and at the same speed as humans, so that they will easily fit in with existing production operations. Let us now take a brief look at each member of the PUMA family.

**250 Series** — The smallest of the PUMA robot family, the 250, shown in Figure 1-25, has the greatest speed and repeatability. Six degrees of freedom combined with an advanced controller make this unit ideal for medium to high-speed assembly and material handling operations. Light and compact, the 250 series provides the versatility to be rapidly integrated into changing work environments. Its size gives it the capability to either stand on its own or become the flexible arm of an automated assembly system. Some typical applications are:

- PC board assembly
- Tester loading
- Adhesive application
- Palletizing material
- Component insertion
- Machine loading
- Material inspection (using optional sight)

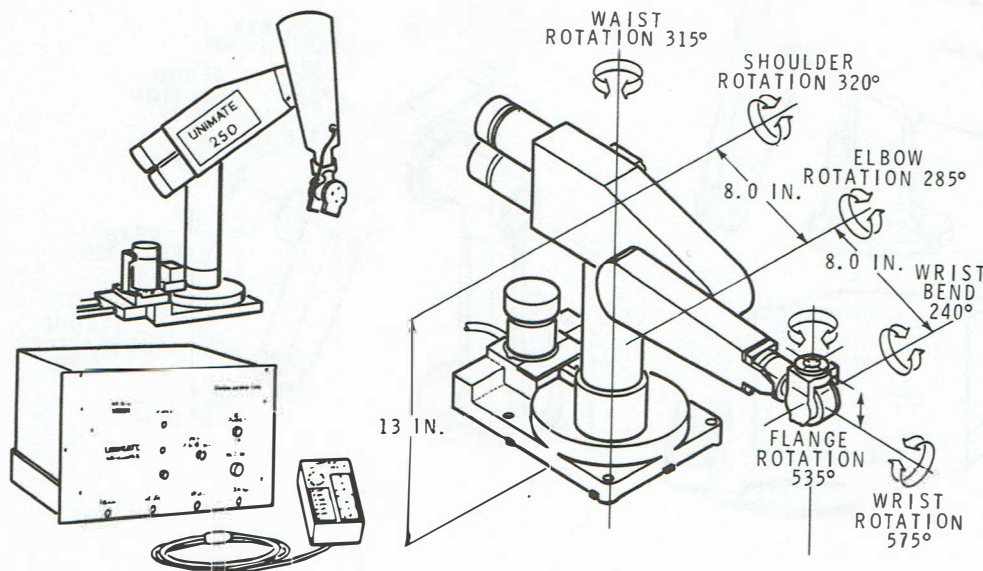


Figure 1-25  
PUMA 250 high-technology robot.

**500 Series** — The workhorse of the PUMA family, the 500, shown in Figure 1-26, is a five-axis robot with the same advanced control system as the 250 series. The manipulator is sized to human dimensions and designed to duplicate human motions. With its 1.0 m reach and heavier load capacity, this robot is capable of handling most industrial assembly, transfer, or packaging operations. While heavier than the 250 series, it retains the same ease of portability. Some of the typical applications are:

- Small part assembly
- Tester loading
- Packaging
- Palletizing
- Parts transfer
- Machine tool loading
- Adhesive application
- Inspection (using optional sight)

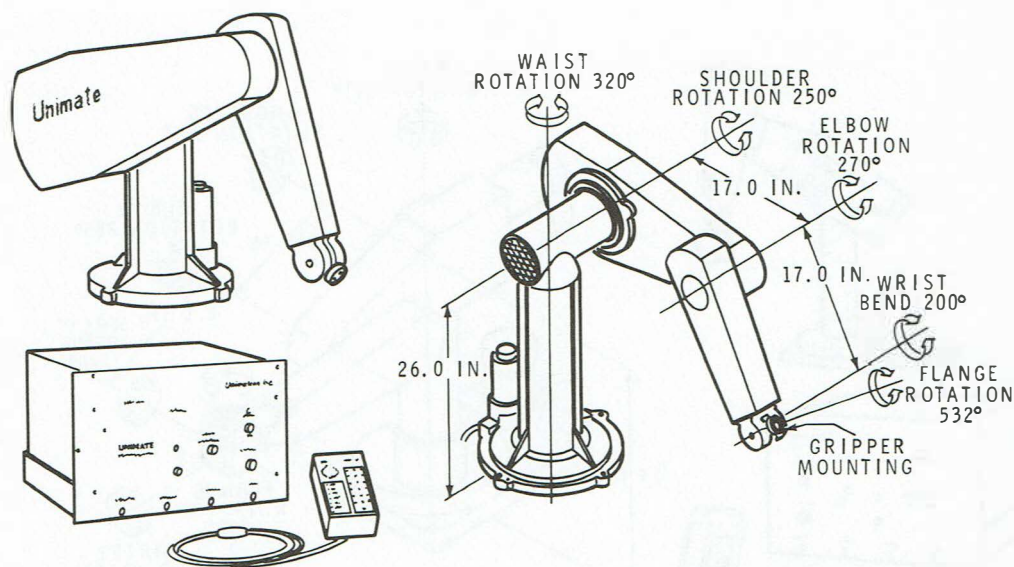


Figure 1-26  
PUMA 500 5-axes robot.

**600 Series** — The most skillful member of the PUMA family, the 600, shown in Figure 1-27, combines the characteristics of both the 250 and 500 series robots and the advanced control system. Utilizing six degrees of freedom and a 1.0 m reach, it has added flexibility to perform complex manipulations within a large working area. The applications for this unit are similar to that of the 500 series.

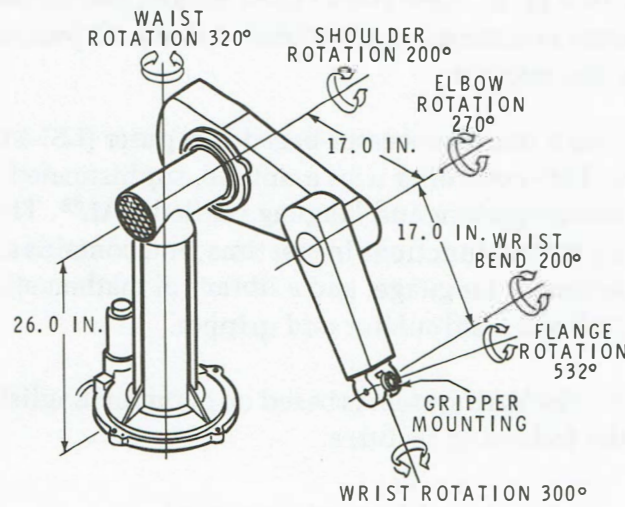


Figure 1-27

PUMA 600 with 6 degrees of freedom.

## GENERAL CHARACTERISTICS

All members of the PUMA robot family are actuated by electric DC servos. Each joint (axis) of the manipulator is driven by a separate microprocessor-controlled DC servo, and has a combined positional accuracy of  $\pm .1$  mm for the 500 and 600 series, and  $\pm .05$  mm for the 250 series. Unlike the simpler robots previously discussed, these high-technology robots can be positioned at any given point along their axis of travel. It is this capability that gives these robots their almost human-like motions. Like many of their counterparts, the PUMA robot uses pneumatic actuation to control the gripper.

PUMA robots use a microprocessor-based computer (LSI-11) to control their functions. This controller uses a unique, sophisticated, high-level, proprietary software system and language called VAL<sup>™</sup>. The VAL multilevel operating system functions in real time, and contains a monitor, an editor, a programming language, and a library of mathematical routines to control the robot's manipulator and gripper.

At the user level, the VAL system is based on common English words and incorporates the following features:

- It operates in a real time environment.
- It is programmable point-to-point and/or continuous path.
- It has an editing capability to add, delete, or replace data.
- It has built-in diagnostic routines.
- It is capable of changing speed.
- Programming interaction can occur during operation (on-line).
- It can accept instructions for looping, branching, and position indexing.
- It has input and output interfaces.
- It is capable of off-line programming.
- It has an off-line floppy disk storage capability.

These features will acquaint you with some of the capabilities of complex controllers. All of these features will be discussed in detail later in the course.

The specifications for the PUMA robot family are shown in Figure 1-28. By comparing these specifications with the specifications of the low and medium-technology robots shown in Figure 1-23, you can easily see the complexity of these high-technology robots.



	250 SERIES	500 SERIES	600 SERIES
	GENERAL:	GENERAL:	GENERAL:
CONFIGURATION	6 revolute axes	5 revolute axes	6 revolute axes
ACTUATION	Electric DC servos	Electric DC servos	Electric DC servos
CONTROLLER	System Computer (LSI-11)	System Computer (LSI-11)	System Computer (LSI-11)
Teaching Method	By manual control and/or computer terminal	By manual control and/or computer terminal	By manual control and/or computer terminal
Program Language	VAL <sup>®</sup>	VAL <sup>®</sup>	VAL <sup>®</sup>
External Program Storage	8K RAM user memory std. (16K RAM max. — optional)	8K RAM user memory std. (16K RAM max. — optional)	8K RAM user memory std. (16K RAM max. — optional)
GRIPPER CONTROL	Floppy-disk (optional)	Floppy-disk (optional)	Floppy-disk (optional)
POWER REQUIREMENT	4-way pneumatic solenoid	4-way pneumatic solenoid	4-way pneumatic solenoid
OPTIONAL ACCESSORIES	110 - 130 VAC, 50 - 60 Hz, 1500 W CRT or TTY terminals, floppy disk memory storage I/O module (8 input/8 output signals — isolated AC/DC levels) Pneumatic gripper w/o fingers (clamping force = 31 lb, finger travel = 0.75 in, flow rate = 1 CFM, air filtered and lubricated)	110 - 130 VAC, 50 - 60 Hz, 1500 W CRT or TTY terminals, floppy disk memory storage I/O module (8 input/8 output signals — isolated AC/DC levels) Pneumatic gripper w/o fingers (clamping force = 31 lb, finger travel = 0.75 in, flow rate = 6 CFM, air filtered and lubricated)	110 - 130 VAC, 50 - 60 Hz, 1500 W CRT or TTY terminals, floppy disk memory storage I/O module (8 input/8 output signals — isolated AC/DC levels) Pneumatic gripper w/o fingers (clamping force = 31 lb, finger travel = 0.75 in, flow rate = 6 CFM, air filtered and lubricated)
<b>PERFORMANCE:</b>			
REPEATABILITY	±0.05 mm (0.002 in)	±0.1 mm (0.004 in)	±0.1 mm (0.004 in)
LOAD CAPACITY (PAYLOAD)	2.0 lb	At flange rotation, 5.5 lb (4 in dia.) (0.5 in-oz-sec <sup>2</sup> ) At wrist bend, 5.5 lb at 5 in (5.7 in-oz-sec <sup>2</sup> )	At flange rotation, 5.5 lb (4 in dia.) (0.5 in-oz-sec <sup>2</sup> ) At wrist bend, 5.5 lb at 5 in (5.7 in-oz-sec <sup>2</sup> )
STRAIGHT LINE VELOCITY	1.0 m/s max.	0.5 m/s (20 in/s) max.	0.5 m/s (20 in/s) max.
ENVIRONMENTAL REQUIREMENTS	10 - 50° C (50 - 120° F) 80% humidity (non-condensing) Shielded against industrial line fluctuations and human electrostatic discharge.	10 - 50° C (50 - 120° F) 80% humidity (non-condensing) Shielded against industrial line fluctuations and human electrostatic discharge	10 - 50° C (50 - 120° F) 80% humidity (non-condensing) Shielded against industrial line fluctuations and human electrostatic discharge
<b>PHYSICAL CHARACTERISTICS:</b>			
ARM WEIGHT	15 lb	120 lb	120 lb
CONTROLLER SIZE	0.48 × 0.32 × 0.51 m (19 in rack mountable)	0.48 × 0.32 × 0.51 m (19 in rack mountable)	0.48 × 0.32 × 0.51 m (19 in rack mountable)
CONTROLLER WEIGHT	80 lb	80 lb	80 lb
CONTROLLER CABLE LENGTH	4.5 m (15 ft) max.	4.5 m (15 ft) max.	4.5 m (15 ft) max.

Figure 1-28  
Specifications for PUMA family.



## Other High-Technology Robots

The PUMA robots are by no means the only high-technology robots being used in industry today. High-technology robots range from a size comparable to the smaller PUMA series 250, which can carry a payload of less than 1 kg, to a much larger robot with the capability of handling over 900 kg. When used with external optional equipment, many of today's robots have the capability to see, feel, hear, and speak. Many of these state-of-the-art techniques will be discussed later in this course.

## Programmed Review

30.	Most medium-technology robots have _____ degrees of freedom than simple robots. (more/less)
31.	(more) The amount of time a machine requires to perform a specific operation directly affects the _____ time of the robot serving that machine.
32.	(cycle) Two factors that greatly affect a robot's accuracy are _____ and _____.
33.	(speed, payload) Most medium-technology robots are _____ actuated.
34.	(hydraulically/electro-hydraulically) The _____ low/medium technology robot does not have the capability to learn.
35.	(low) The "rotating drum" controller _____ (can/cannot) be used to store a specific program.
36.	(can) The _____ robots are considered to be the elite members of the robot family.
37.	(high-technology) Since high-technology robots can work at the same _____ as humans, they fit easily into existing production operations.
38.	(speed) Most high-technology robots use a _____ for control.
39.	(computer/microprocessor) The _____ robot has the capability of carrying the greatest payload.
	(high-technology)

## EXPERIMENTS

Perform Robot Familiarization Experiments 1 and 2. You will find these experiments in Unit 12. After you have finished these experiments, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct ones that appear after the exam.

1. The major difference between hard automation and robots is that robots:
  - A. Work faster.
  - B. Require less maintenance.
  - C. Can be programmed for different tasks.
  - D. Can handle greater payloads.
2. Which of the following device(s), when added to hard automation equipment, helped bring about the first simple robots?
  - A. Speed controllers.
  - B. Timers.
  - C. Sequencers.
  - D. All of the above.
3. Which type of control system uses feedback to compare the output signal to the input signal?
  - A. Servo.
  - B. Mechanical.
  - C. Pneumatic.
  - D. Synchro.
4. Which device helped make the special purpose robot economically practical?
  - A. Transistor.
  - B. Microprocessor.
  - C. Computer.
  - D. Hydraulic cylinder.
5. The part of the robot that performs the actual work is called the:
  - A. Controller.
  - B. Manipulator.
  - C. Rotating drum.
  - D. Servo control system.

6. What is considered the brains and nervous system of the robot?
- A. Television camera.
  - B. Controller.
  - C. Actuator.
  - D. Gripper.
7. Which of the following is not a common method of actuating a robot?
- A. Hydraulic and pneumatic cylinders.
  - B. Electric motors.
  - C. Mechanical drives.
  - D. Hydraulic and pneumatic rotary motors.
8. Spray painting and welding operations are best handled by which type of robot?
- A. Medium-technology.
  - B. Limited-sequence.
  - C. Low-technology.
  - D. High-technology.
9. Which component provides the energy used to position the robot's manipulator?
- A. Power supply.
  - B. Sequencing valve.
  - C. Controller.
  - D. Servo control system.
10. What determines the number of intricate motions a robot can perform?
- A. Type of power supply used.
  - B. Number of axes.
  - C. Type of actuation used.
  - D. Cycle time.
11. Which category of robot is best suited for machine loading/unloading of medium weight objects?
- A. Low-technology.
  - B. Puma series 250.
  - C. Medium-technology.
  - D. High-technology.

12. Which of the following is not a characteristic of pneumatic actuation devices?
- A. Low cost.
  - B. High speed.
  - C. High payloads.
  - D. Readily available source of power.
13. The basic difference between non-servo controlled and servo-controlled robots is the:
- A. Type of controller used.
  - B. Number of degrees of freedom.
  - C. Type of actuation used.
  - D. Amount of payload they can handle.
14. Which type of control allows the robot to smoothly follow a particular path?
- A. Continuous path servo control.
  - B. Point-to-point servo control.
  - C. Continuous path and point-to-point servo control.
  - D. Mechanical control.
15. Which robot has a work envelope that is a small portion of a sphere?
- A. Jointed-spherical coordinate robot.
  - B. Cylindrical coordinate robot.
  - C. Jointed-cylindrical coordinate robot.
  - D. Spherical coordinate robot.
16. What component of the robot has the task of positioning the robot arm to a pre-selected point?
- A. Gripper.
  - B. Actuator.
  - C. Manipulator.
  - D. End stops.
17. The most common method of controlling a high-technology robot is:
- A. Computer/Microprocessors.
  - B. Programmable rotating drums.
  - C. Mechanically set computer controlled end stops.
  - D. Air-logic programmers.



18. Low-technology robots are best suited for which type of task?
- A. Welding operations.
  - B. Palletizing material.
  - C. Material handling.
  - D. Forging.
19. What factor(s) has the greatest effect on how fast a robot must perform a certain task?
- A. Distance of travel and amount of payload.
  - B. Type of actuation and controller used.
  - C. Speed and cycle time of the robot.
  - D. Cycle time of the machine it serves.
20. Which category of robot has the greatest degree of accuracy?
- A. Medium-technology.
  - B. High-technology.
  - C. Low-technology.
  - D. They all have the same degree of accuracy.

## EXAMINATION ANSWERS

For your convenience, the page where the correct answer can be found is shown following the answer.

1. C — Can be programmed for different tasks. [ 1-7 ]
2. D — All of the above. [ 1-7 ]
3. A — Servo. [ 1-7, 1-14 ]
4. B — Microprocessor. [ 1-7 ]
5. B — Manipulator. [ 1-12 ]
6. B — Controller. [ 1-12 ]
7. C — Mechanical drives. [ 1-12 ]
8. D — High-technology. [ 1-17 ]
9. A — Power supply. [ 1-12 ]
10. B — Number of axes. [ 1-9 ]
11. C — Medium-technology. [ 1-33, 1-34 ]
12. C — High payloads. [ 1-25, 1-37 ]
13. B — Number of degrees of freedom. [ 1-14 ]
14. A — Continuous path servo control. [ 1-14 ]
15. D — Spherical coordinate robot. [ 1-13 ]
16. B — Actuator. [ 1-12 ]
17. A — Computer/Microprocessors. [ 1-40, 1-44 ]

18. C — Material handling. [ 1-17 ]
19. D — Cycle time of the machine it serves. [ 1-23 ]
20. C — Low-technology. [ 1-22, 1-24 ]

*Unit 2*

**AC AND FLUIDIC POWER**

## CONTENTS

Introduction .....	2-3
Unit Objectives .....	2-4
Unit Activity Guide .....	2-5
AC Power Generation .....	2-6
AC Motors .....	2-22
Basic Hydraulic System .....	2-43
Pneumatic Systems .....	2-62
Experiment .....	2-66
Unit Examination .....	2-67
Examination Answers .....	2-73



## INTRODUCTION

As you recall, the “walking locomotive”, briefly discussed in Unit One, was a mechanical-type man powered by a small steam engine. Steam was a primary method of producing power during this period of time, but it had many drawbacks. Steam-generated power wasn’t instantaneous — first, a fire had to be built and water heated; it was difficult to regulate and control; and it was severely restricted in usage due to its size, weight, and constant demand for fuel and water. Steam is still a major source of power today, but now it is used in a more conventional form; large steam turbines are used to drive AC generators which, in turn, produce electrical power for home, farm, and industry. AC power, the most common type of power used today, is instantaneous, easy to regulate and control, and, with today’s modern distribution systems, available virtually everywhere. Even if commercial AC power isn’t readily available, small, engine-driven AC generators are available.

AC power is used in almost every aspect of industry. In the field of robotics, AC power is used to drive numerous electric motors for robot positioning. Even when robots are hydraulically or pneumatically actuated, AC power is required to drive the associated pumps and compressors. In addition, AC power is also widely used, once it has been converted to DC, in the robot’s sensing and control circuits. Without power, a robot is just an inert mass of gears and cams, pistons and valves, switches and relays; plus diodes, transistors, and numerous other electronic devices. However, once power has been provided to this lifeless assemblage of mechanical and electrical components, it becomes a valuable machine, dedicated to serving man. In this unit you will learn how AC power is generated and ultimately used to power AC motors and fluid power systems. A fluid power system can be either a hydraulic system, using oil; or a pneumatic system, using air; where generated pressure is the prime moving force.

Examine the “Unit Objectives” listed in the next section to see what you will learn in this unit. Then follow the instructions in the “Unit Activity Guide” to be sure you perform all the steps necessary to complete this unit successfully. Check off each step as you complete it and, in the spaces provided, keep track of the time you spend on each activity.

## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. State the factors that affect the rating of an AC generator.
2. Explain how a load is connected to a three-phase, wye-configured, AC generator.
3. State the major differences between a wye-connected and a delta-connected three-phase generator.
4. Explain how a revolving field is accomplished in a three-phase AC motor.
5. Explain how a revolving field is developed in a single-phase capacitor motor.
6. Describe how you reverse direction in a three-phase induction motor.
7. State the difference between a cage rotor and form-wound rotor.
8. Explain the function of the basic components of a hydraulic system.
9. Explain the function of the basic components of a pneumatic system.
10. Describe how three-phase AC generators operate.
11. Describe the operation of a three-phase induction motor.
12. Explain the operation of a split-phase motor.
13. Explain the operation of a capacitor-start AC motor.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "AC Power Generation."	_____
<input type="checkbox"/> Answer Programmed Review Questions 1-14.	_____
<input type="checkbox"/> Read "AC Motors."	_____
<input type="checkbox"/> Answer Programmed Review Questions 15-27.	_____
<input type="checkbox"/> Read "Basic Hydraulic System."	_____
<input type="checkbox"/> Answer Programmed Review Questions 28-38.	_____
<input type="checkbox"/> Read "Pneumatic Systems."	_____
<input type="checkbox"/> Answer Programmed Review Questions 39-44.	_____
<input type="checkbox"/> Perform Experiment 3.	_____
<input type="checkbox"/> Complete The Unit Examination.	_____
<input type="checkbox"/> Check The Examination Answers.	_____

## AC POWER GENERATION

Most electric power utilized by home, farm, or industry is generated by alternating current generators; they are also widely used in aircraft and automobiles. AC generators are made in many different sizes, depending on their intended use. For example, the large hydro-electric generator, shown in Figure 2-1, can produce millions of volt-amperes for distribution over a wide area; while smaller generators, like those used on aircraft, produce only a few thousand volt-amperes.

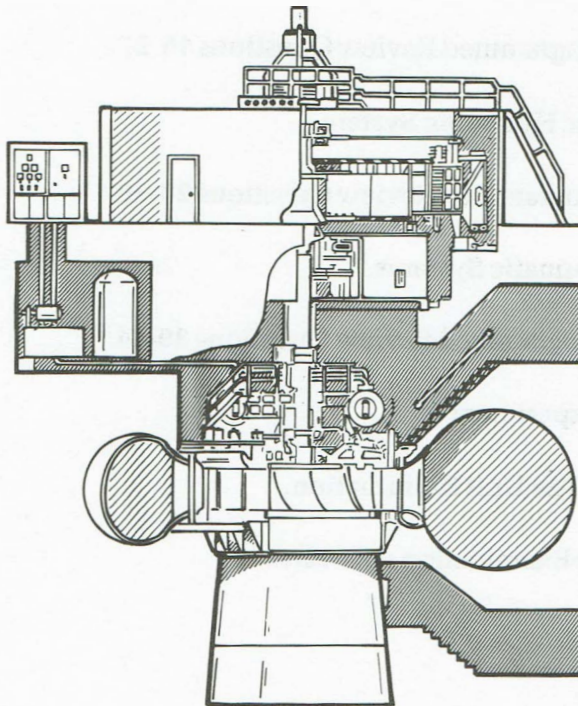


Figure 2-1  
Vertical Hydroelectric Generator  
(output approximately 13,500 VAC)

Virtually all robots used in industry are considered to be stationary robots. This means that they are physically mounted in a specific area and have a limited amount of freedom to move about. Granted, some industrial robots are mounted on tracks or guides, giving them a somewhat greater freedom of movement, but this is the exception rather than the rule. Because of this restricted freedom, providing power to an industrial robot is relatively simple. The power required to operate various industrial robots can vary greatly. This can be seen from Figure 2-2.

ROBOT	POWER REQUIREMENTS
Cincinnati Milacron Heavy Duty HT <sup>3</sup>	230/460 volts, 3 phase, 60 Hz, 32 kVA
Unimate 1000	460 volts, 3 phase, 60 Hz, 9 kVA
Unimate 2000B	460 volts, 3 phase, 60 Hz, 11 kVA
Unimate 4000B	460 volts, 3 phase, 60 Hz, 34 kVA
Auto-place AP50	Shop Air @ 80 PSI, 115 volts, 60 Hz, 2 amps
Thermwood series 3 General purpose robot	240/480 volts, 3 phase, 60 Hz, 5 kW
Puma 250 series	95 - 130 volts, 1 phase, 50 - 60 Hz, 750 VA max.
Model 600 controller for PRAB/VERSATRAN series F robots	220/440 volts, 3 phase, 50 - 60 Hz, 100 amps max.
Unimation apprentice robot	240/480 volts +10% - 15%, single phase, 60 Hz, 1 kVA
Nordson robot	230/460 volts, 3 phase, 60 Hz.

Figure 2-2  
Typical Robot Power Requirements



## **AC Generator Characteristics**

Regardless of size, all generators operate on the same basic principle — a magnetic field cutting through conductors, or conductors passing through a magnetic field. Thus, generators have at least two distinct sets of conductors. They are (1) a group of conductors in which the output voltage is generated, and (2) a second group of conductors through which direct current is passed to obtain an electromagnetic field of fixed polarity. The conductors in which the output voltage is generated are always referred to as the armature windings. The conductors in which the electromagnetic field originates are always referred to as the field windings.

In addition to the armature and the field, there must also be motion between the two. To provide this, AC generators are built in two major assemblies, the stator and the rotor. The rotor rotates inside the stator. The rotor may be driven by any one of a number of commonly used power sources, such as steam or water-driven turbines, internal-combustion engines, and even large electric motors.

## **Types of AC Generators**

There are two basic types of alternating current generators in use today: the revolving armature type and the revolving field type. However, they both perform the same basic function.

### **REVOLVING ARMATURE TYPE GENERATOR**

In the revolving-armature AC generator, the stator provides a stationary electromagnetic field. The rotor, acting as the armature, revolves in the field, cutting the lines of force, producing the desired output voltage. In this generator, the armature output is taken through slip rings and thus retains its alternating characteristics. For a number of reasons, the revolving-armature AC generator is seldom used. Its primary limitation is the fact that its output power is conducted through sliding contacts (slip rings and brushes). These contacts are subject to frictional wear and sparking. In addition, they are exposed, and thus liable to arc-over at high voltages. Consequently, revolving-armature generators are restricted to low-power, low-voltage applications and are not used where large amounts of power and voltage are required, such as a commercial generating plant.

## REVOLVING FIELD TYPE GENERATOR

The revolving-field AC generator, shown in Figure 2-3, is by far the most widely used type. In this type of generator, direct current from a separate source is passed through the windings of the rotor by means of slip rings and brushes. This action produces a rotating electromagnetic field of fixed polarity. The rotating magnetic field, following the rotor, extends outward and cuts through the armature windings imbedded in the surrounding stator. As the rotor turns, alternating voltages are induced in the windings since magnetic fields of first one polarity and then the other cut through them. Since the output power is taken from stationary windings, the output may be connected through fixed terminals T1 and T2 in Figure 2-3. This is advantageous, in that there are no sliding contacts; and the whole output circuit is isolated, thus minimizing the danger of arc-over.

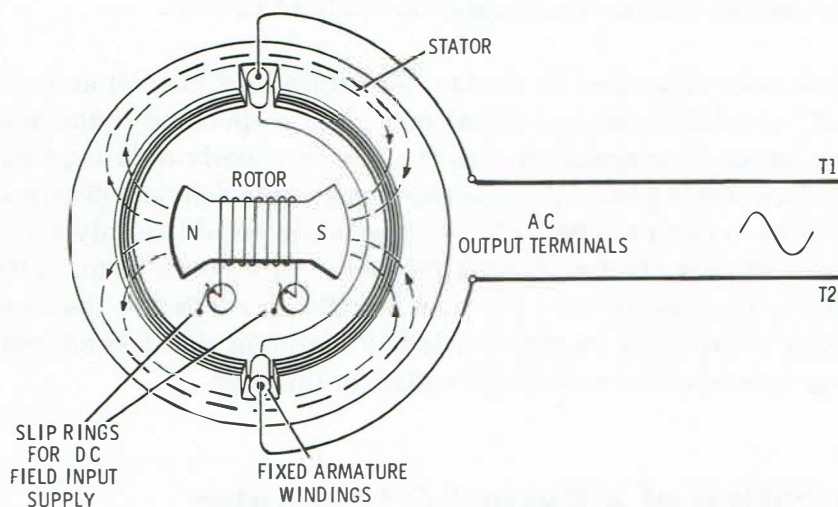


Figure 2-3  
Revolving-Field AC Generator

Slip rings and brushes are adequate for the DC field supply because the power level in the field is much smaller than in the armature circuit.

## Rating of AC Generators

The rating of an AC generator pertains to the load it is capable of supplying. The normal-load rating is the load it can carry continuously; while its overload rating is the above-normal load which it can carry for specified lengths of time only. The load rating of a particular generator is determined by the internal heat it can withstand. Since heating is caused mainly by current flow, the generator's rating is identified very closely with its current capacity.

The maximum current that an AC generator can supply depends upon (1) the maximum heating loss ( $I^2R$  power loss) that can be sustained in the armature and (2) the maximum heating loss that can be sustained in the field. In AC generators, current varies with the load; and a lagging power-factor — the ratio of the average power to the apparent power of the load circuit — tends to demagnetize the field. Thus, terminal voltage is maintained only by increasing the DC field current.

AC generators are rated in terms of armature load current and voltage output, or kilovolt-ampere (kVA) output, at a specified frequency and power factor. The specified power factor is usually 80% lagging. For example, a single-phase AC generator designed to deliver 100 amperes at 1000 volts is rated at 100 kVA. This generator would supply a 100 kW load at unit power factor or an 80 kW load at 80% power factor. If the AC generator supplied a 100 kVA load at 20% power factor, the required increase in DC field current needed to maintain the desired terminal voltage would cause excessive heating in the field.

## Operation of a Basic AC Generator

A typical rotating-field AC generator consists of an AC generator and a smaller DC generator built into a single unit. The output of the AC generator section supplies alternating current to the load for which it was designed; while the DC generator's only purpose is to supply the direct current required to maintain the electromagnetic field. The DC generator portion of the unit is referred to as the exciter. A typical AC generator is shown in Figure 2-4A. Figure 2-4B is a simplified schematic diagram of the generator.

The rotary force used to drive the generator is transmitted through the rotor drive shaft (1) (Figure 2-4A). The exciter field (2), shown in Figure 2-4B, creates an area of intense magnetic flux between its poles. When the

exciter armature (3) is rotated in the exciter field flux, voltage is induced into the exciter armature windings. The exciter output commutator and brushes (4) connect the exciter output directly to the AC generator field input slip rings and brushes (5). Since these slip rings, rather than a commutator, are used to supply current through the AC generator field (6), current always flows in one direction through these windings. Thus, a fixed polarity magnetic field is maintained at all times in the AC generator field windings. When the AC generator field is rotated, its magnetic flux is passed through and across the AC generator armature windings (7). Remember, a voltage is induced into a conductor if it is stationary and a magnetic field is passed across the conductor, the same as if the field is stationary and the conductor is moved. The alternating voltage induced in the AC generator armature windings is connected through fixed terminals to the load.

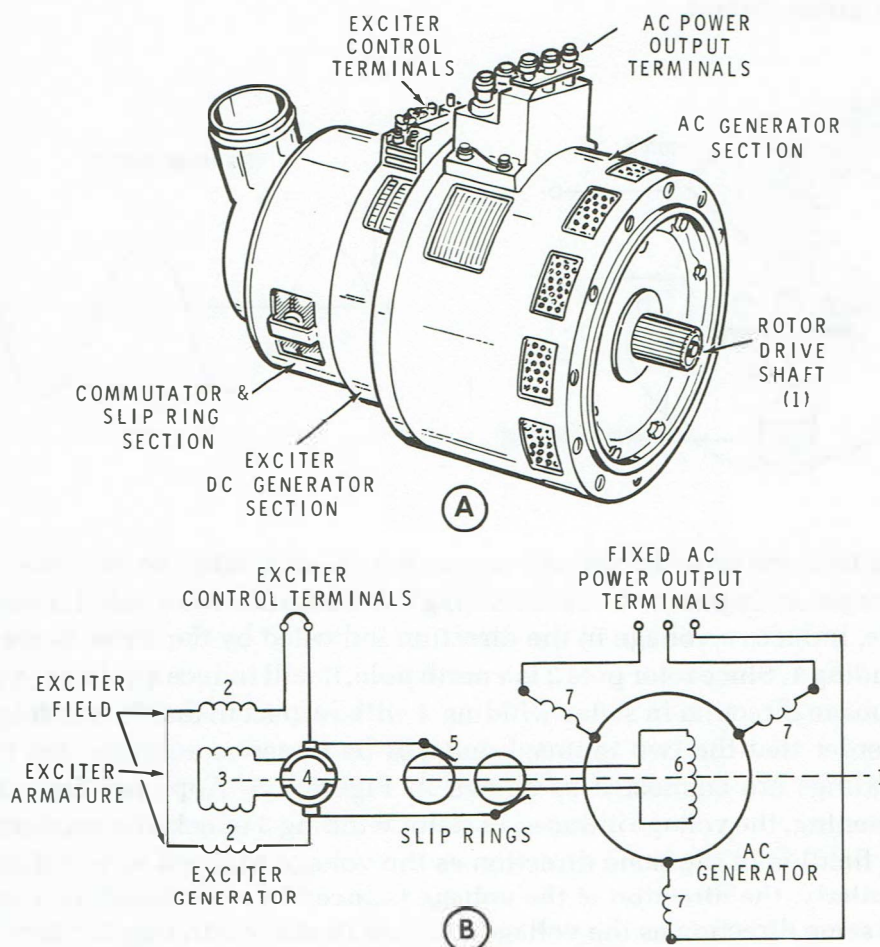


Figure 2-4

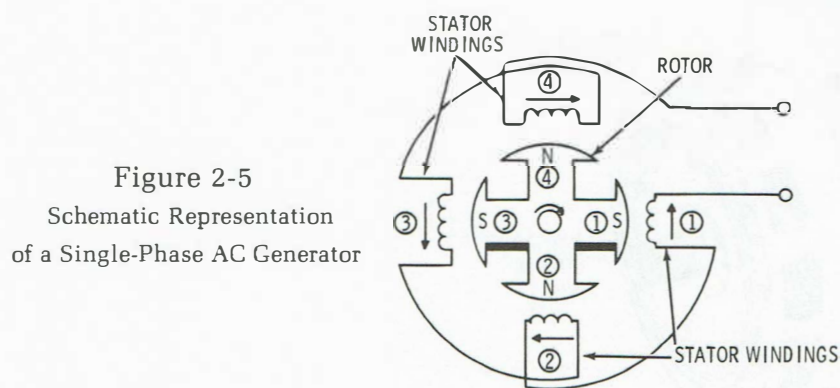
(A) Typical Rotating-Field AC Generator  
 (B) Simplified Schematic of 3-Phase AC Generator



## Single-Phase Generators

A single-phase AC generator has a stator made up of a number of windings in series, which form a single circuit in which an output voltage is generated.

Figure 2-5 illustrates a schematic diagram of a single-phase AC generator that has four poles. The stator has four polar groups evenly spaced around the stator frame; while the rotor also has four poles, with adjacent poles of opposite polarity. As the rotor revolves, the AC voltages are induced in the stator windings. Since one rotor pole is in the same position relative to a stator winding as any other rotor pole, all stator pole groups are cut by equal amounts of magnetic lines of force at any given time. As a result, the voltages induced in all the windings have the same amplitude or value at any given instant.



The four stator windings are connected to each other so that the AC voltages are in phase, or “series aiding”. Assume that rotor pole 1, a south pole, induces a voltage in the direction indicated by the arrow in stator winding 1. Since rotor pole 2 is a north pole, it will induce a voltage in the opposite direction in stator winding 2 with respect to that in winding 1. In order that the two induced voltages be in series addition, the two windings are connected as shown in Figure 2-5. Applying the same reasoning, the voltage induced in stator winding 3 (clockwise rotation of the field) is in the same direction as the voltage induced in winding 1. Similarly, the direction of the voltage induced in stator winding 4 is in the same direction as the voltage induced in stator winding 2. Since all four stator winding groups are connected in series, the voltages induced in each winding add; thus, the total voltage produced is four times the voltage in any one winding.

## Three-Phase Generators

The three-phase (polyphase) AC generator, as the name implies, has several sets of three, single-phase windings spaced around the stator in such a manner that the voltage in each winding is  $120^\circ$  out of phase with the voltages in the other two windings of that set. A schematic diagram of a three-phase stator showing all the winding sets becomes complex, and it is difficult to see what is actually happening. Therefore, a simplified schematic diagram, showing all the windings of the single phases (A, B, and C) condensed together as one winding, is shown in Figure 2-6. The rotor is omitted from the schematic diagram for simplicity of explanation. The voltage waveforms of each phase are shown to the right of the schematic. The three voltages are  $120^\circ$  apart and are similar to the voltages that would be generated by three, single-phase AC generators whose voltages are out of phase by angles of  $120^\circ$ . The three phases are independent of each other.

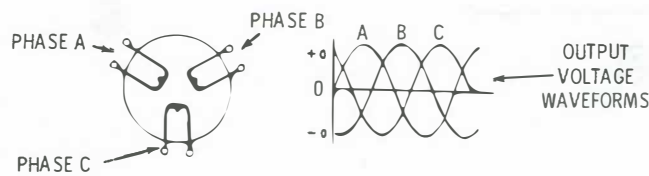


Figure 2-6  
Simplified Schematic of A Three-Phase Generator



## THE WYE CONNECTION

Rather than have six leads come out of the three-phase generator, one of the leads from each phase may be connected to form a common junction. The stator is then called wye, or star-connected. The common lead may or may not be brought out of the generator. If it is brought out, it is called the neutral. The simplified schematic diagram of Figure 2-7A shows a wye-connected stator with the common lead not brought out.

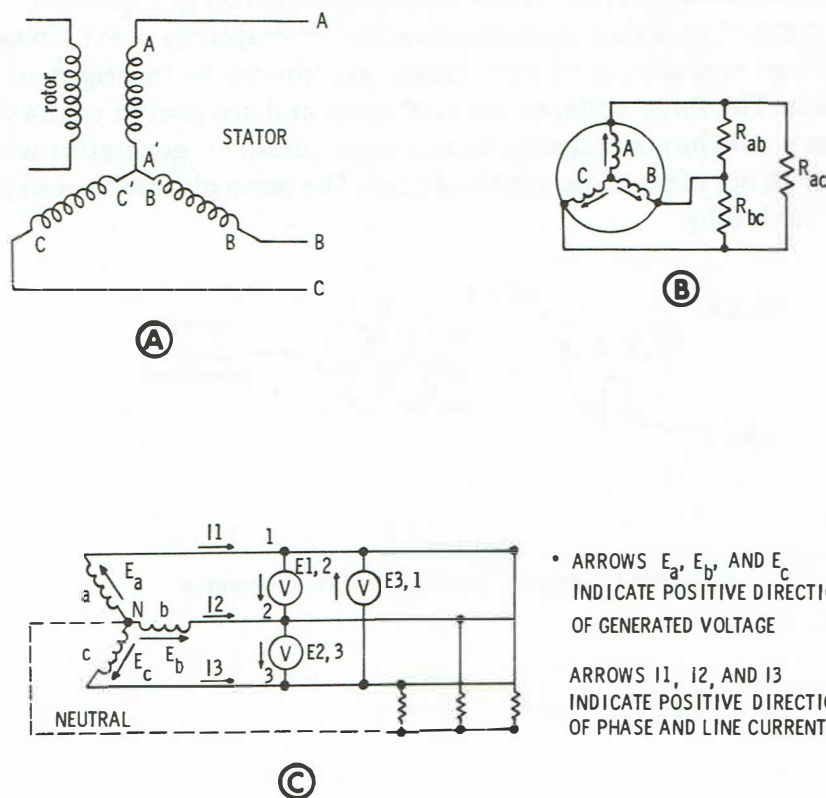


Figure 2-7  
Wye-Connected Three-Phase Generator

Each load is connected across two phases in series, as seen in Figure 2-7B. Thus,  $R_{ab}$  is connected across phases A and B in series,  $R_{ac}$  is connected across phases A and C in series, and  $R_{bc}$  is connected across phases B and C in series. Consequently, the voltage across each load is larger than the voltage across a single phase. In a wye-connected generator the three start ends of each single-phase winding are connected together to a common neutral point, and the opposite, or finish, ends are connected to the line terminals A, B, and C. These letters are always used to designate the three phases of a three-phase system, or the three line wires to which the AC generator phases connect.

A three-phase wye-connected AC generator supplying three separate loads is shown in Figure 2-7C. When unbalanced loads are being supplied, a neutral may be added, as shown in the figure by the broken line between the common neutral point and the loads. The neutral wire serves as a common return circuit for all three phases and maintains a voltage balance across the loads. No current flows in the neutral wire when the loads are balanced. This system is known as a 3-phase 4-wire circuit and is widely used to distribute 3-phase power in industrial settings.

The line voltage is greater than the voltage of a single phase in the wye-connected circuit because there are two phases connected in series between each pair of line wires, and their voltages combine. Line voltage is not twice the value of a single phase voltage, however, because the phase voltages are not in phase with each other. Therefore, the wye-connection is used where high voltage outputs are desired; with the output voltage being  $\sqrt{3}$ , or approximately 1.73 times as great as single phase voltage.

## THE DELTA CONNECTION

A three-phase stator may also be connected as shown in Figure 2-8A. This configuration is called the delta connection. In a delta-connected AC generator, as shown in Figure 2-8B, the start end of the first phase winding is connected to the finish end of the third; the start end of the third phase winding is connected to the finish of the second phase winding; and the start of the second phase winding is connected to the finish of the first phase winding. The three junction points are connected to the line wires leading to the load.

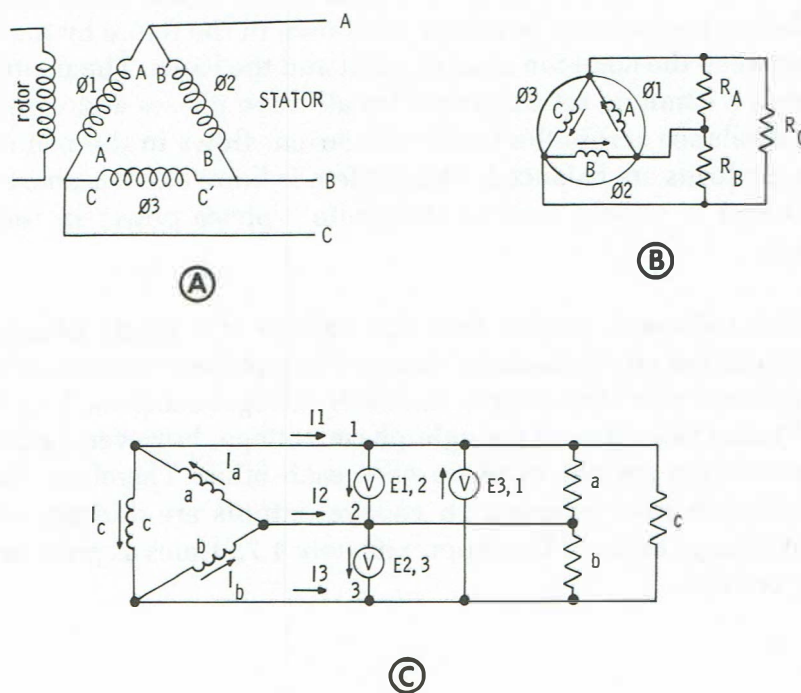


Figure 2-8  
Delta-Connected Three-Phase Generator

A three-phase delta-connected AC generator is depicted in Figure 2-8C. The generator is connected to a 3-phase 3-wire circuit which supplies a 3-phase load. Because the loads are connected directly across the phases, line voltage is equal to phase voltage. When the generator phases are properly connected in delta, no appreciable current flows within the delta loop when there is no external load connected to the generator. If any one of the phases is reversed with respect to its correct connection, a short-circuit current flows within the windings at no load, causing damage to the windings.

To avoid connecting a phase in reverse, you have to test the circuit before closing the delta. You can do this by connecting a voltmeter between the two ends of the delta loop before you close the delta. The two ends of the delta loop should never be connected if there is any indication of an appreciable current or voltage between them when no load is connected to the generator.

In a delta-connected generator, there is more than one internal path for current to flow toward the load. This results in a higher current-carrying capacity for the generator. Therefore, line current is greater than phase current, but not twice as great because the phase currents are not in phase with each other. Line current is approximately 1.73 times as great as single-phase current in a delta-connected generator.

Thus, to summarize the major difference between delta and wye configured generators:

#### WYE

$$E_{Line} = \sqrt{3} \times E_{phase}$$

$$I_{Line} = I_{phase}$$

#### DELTA

$$E_{Line} = E_{phase}$$

$$I_{Line} = \sqrt{3} \times I_{phase}$$

## AC Generator Considerations

The following considerations must be taken into account when using AC generators:

### SINGLE AND THREE PHASE VOLTAGE OUTPUTS

When it is necessary to supply both single-and three-phase voltages from a single three-phase AC generator, the wye connection, as shown in Figure 2-9, is used. By connecting the neutral lead to the common junction it becomes possible to obtain three single-phase voltages and one three-phase voltage. For example, from a generator designed to produce a 208 volt, three-phase, 60 Hz output, it would be possible to obtain three additional single-phase voltages of 120 volts each by using the four-wire wye connection. To keep this type of generator from overheating, it is important that the load be distributed evenly across the three windings.

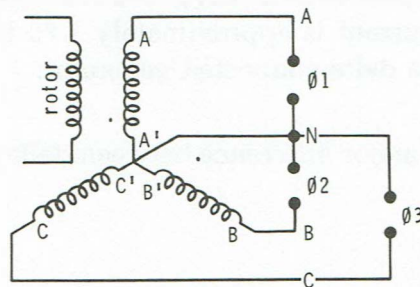


Figure 2-9  
Three-Phase Wye-Connected Generator Supplying  
3 Single-Phase Outputs.

## REGULATION OF AC GENERATORS

In a single-phase AC generator with one pair of poles, the rotor must complete one revolution to produce one complete cycle of alternating current. Thus, if the number of pairs of poles were increased, the output frequency would also increase with no increase in speed of rotation required. This is shown in the following formula:

$$f = SP/60$$

where  $f$  = frequency, Hz

$S$  = speed of generator, rpm

$P$  = number of pairs of poles

60 = conversion factor ( $f$  in Hz and  $S$  in rpm)

If additional pairs of poles are added, the frequency will increase in steps, whereas an increase in speed of generator rotation will result in a comparatively smooth increase in frequency. Hence, the number of pairs of poles is determined by the frequency for which the generator is designed, and small corrections in frequency are made by varying the speed of rotation.

The value of the generator output voltage is determined by the speed of rotation, number of conductors, and the strength of the flux field being cut. Since the speed of rotation will also effect the frequency output, it is not practical to regulate the voltage output by varying speed. Also, the number of conductors used is a design consideration and cannot be varied. Therefore, to regulate output voltage, the DC exciter current is varied. This varies the field flux of the exciter unit and thus changes the amplitude of the induced voltage.



## Programmed Review

1. All AC generators work on the principle of a \_\_\_\_\_ cutting through conductors.
2. (magnetic field) The two major assemblies of an AC generator are the \_\_\_\_\_ and the \_\_\_\_\_.
3. (stator, rotor) In the revolving-\_\_\_\_\_ AC generator, direct current from a separate source is passed through the windings of the rotor.
4. (field) The \_\_\_\_\_ load rating of an AC generator is determined by the load it can supply continuously.
5. (normal) In a rotating-field AC generator, a magnetic field of \_\_\_\_\_ polarity is maintained at all times.
6. (fixed) The DC \_\_\_\_\_ portion of a rotating-field AC generator is called the exciter.
7. (generator) In a single-phase AC generator that has four groups of series aiding windings, the total output voltage of the generator would be \_\_\_\_\_ times as great as the output of a single group of windings.
8. (four) In a three-phase AC generator, the voltage of any one winding or group of windings is \_\_\_\_\_ degrees out of phase with the other two.
9. (120) In a three-phase AC generator, the stator is said to be \_\_\_\_\_ connected when one lead from each phase is connected to a common junction.

10. (wye or star) When no current flows in the neutral wire of a three-phase wye connected generator, it can be assumed that the loads are in a \_\_\_\_\_ condition.

11. (balanced) In a three-phase delta-connected AC generator, the finish end of one winding is connected to the \_\_\_\_\_ end of another winding.

12. (start) In a \_\_\_\_\_ connected AC generator there is more than one internal path for current to flow.

13. (delta) If the number of pairs of poles of an AC generator were increased and the speed of rotation remained the same, the output \_\_\_\_\_ would also increase.

14. (frequency) Increasing the speed of rotation of an AC generator \_\_\_\_\_ the recommended method of increasing the  
(is/is not)  
output voltage.

(is not)

## AC MOTORS

Most of the power generating systems produce alternating current. However, there are other advantages in the use of AC motors besides the wide availability of AC power. In general, AC motors are less expensive than DC motors. Most types of AC motors do not employ brushes and commutators. This eliminates many problems of maintenance and wear. In addition, it eliminates the problem of dangerous arcing.

DC motors, which will be discussed in detail in Unit Three, are best suited for some uses, such as applications that require variable-speed motors. However, in the great majority of applications, the AC motor is best; and the design and increasing use in recent years of variable-speed controllers for induction motors has greatly increased the use of AC motors where speed control is required.

AC motors are manufactured in many different sizes, shapes, and ratings for use on an even greater number of jobs. They are designed for use with either single or polyphase systems.

Since this unit cannot possibly cover all aspects of the subject of AC motors, we will restrict ourselves to the more common types — the rotating-field induction motor and the synchronous motor.

## Rotating Field

Before we go into how a rotating magnetic field will cause an energized rotor to turn, let's take a few minutes to find out how a rotating magnetic field is produced.

The rotating field is set up by out-of-phase currents in the motor's stator windings. Figure 2-10 illustrates the manner in which a rotating field is produced by stationary coils, or windings, when they are supplied by a 3-phase current source. For ease of explanation, rotation of the field is developed in Figure 2-10 by "stopping" it at six selected positions, or instants. These instants are marked off at  $60^\circ$  intervals on the sine waves representing currents in the three phases, A, B, and C.

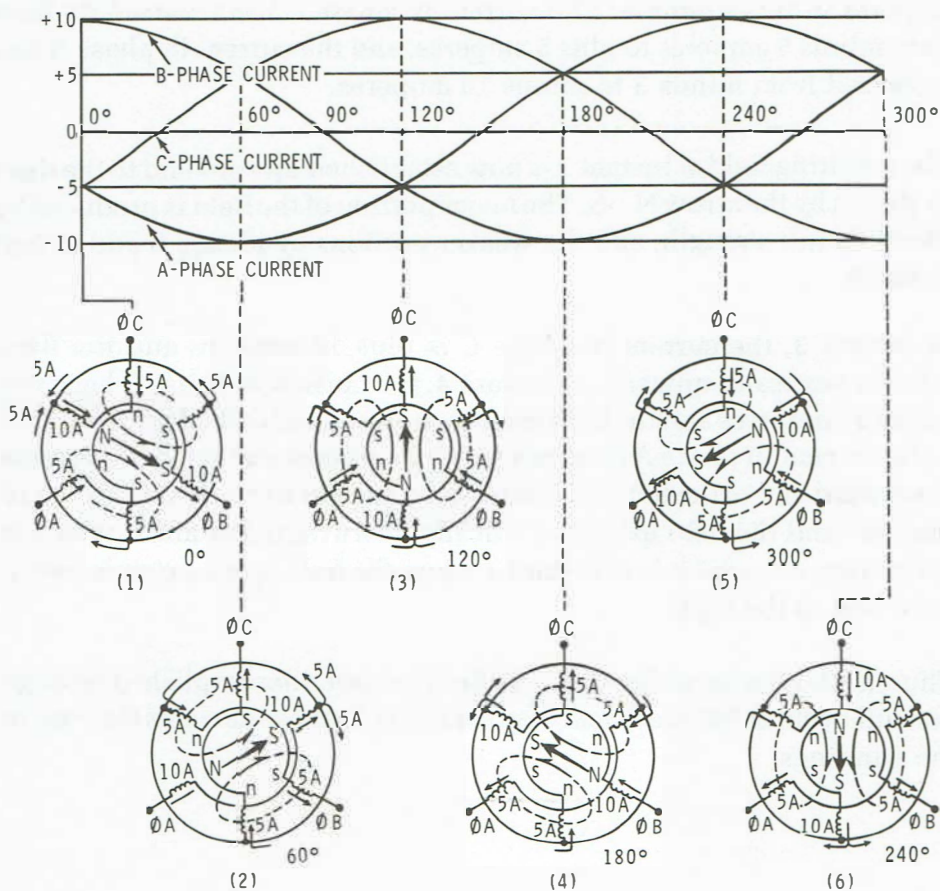


Figure 2-10  
Rotating Magnetic Field

At instant 1, the current in phase B is maximum positive; assume plus 10 amperes in this example. Current is considered to be positive when it is flowing out from a motor terminal, and negative when it flows into a motor terminal. At the same time (instant 1), current flows into the A and C terminals at half value, minus 5 amperes each in this case. These currents combine at the neutral, or common, connection to supply plus 10 amperes out through the B phase.

The resulting field at instant 1 is established downward and to the right as shown by the arrow  $N \rightarrow S$ . The major portion of this field is produced by the B phase, full strength at this time, and is aided by the adjacent phases A and C, which are half strength. The weaker portions of the field are indicated by the letters "n" and "s." This field is a two-pole field extending across the space that would normally contain the rotor.

At instant 2, the current in phase B is reduced to half value, plus 5 amperes in this example. The current in phase C has reversed its flow from minus 5 amperes to plus 5 amperes, and the current in phase A has increased from minus 5 to minus 10 amperes.

The resulting field at instant 2 is now established upward and to the right as shown by the arrow  $N \rightarrow S$ . The major portion of the field is produced by phase A, full strength, and the weaker portions by phases B and C, half strength.

At instant 3, the current in phase C is plus 10 amperes and the field extends vertically upward. At instant 4, the current in phase B becomes minus 10 amperes and the field extends upward and to the left. At instant 5, the current in phase A becomes plus 10 amperes and the field extends downward and to the left. At instant 6, the current in phase C is minus 10 amperes and the field extends vertically downward. Instant 7, which is not shown, corresponds to instant 1 when the field again extends downward and to the right.

Thus, a full rotation of the two-pole field has been accomplished through one full cycle of 360 electrical degrees of the 3-phase currents flowing in the windings.



The direction of rotation of the magnetic revolving field would be changed if you interchanged any two line leads to the three motor terminals. Consider Figure 2-11 for example. If line 1 connects to phase A, line 2 to phase B, and line 3 to phase C, and if the line currents reach their positive maximum values in the sequence 1, 2, 3, the phase sequence A, B, C and the rotation is arbitrarily clockwise. If lines 1 and 2 are interchanged, the phase sequence becomes B, A, C, and the revolving field turns counterclockwise.

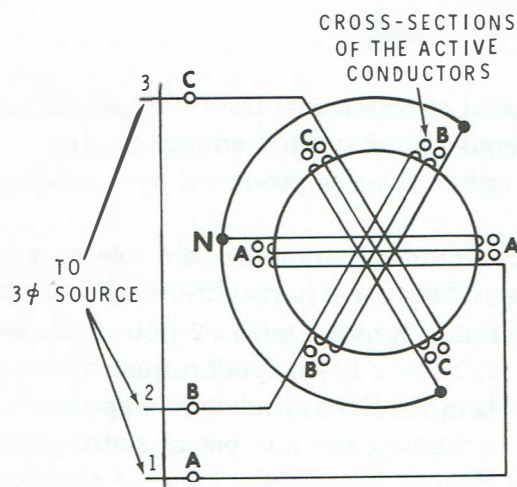


Figure 2-11  
3-Phase Induction Motor Windings

Most induction motors utilized today are designed to operate on single-phase power, or the three-phase supply used in the preceding discussion. The out-of-phase currents necessary to produce a rotating field are inherent in the three-phase supply, since 3-phase voltages are generated  $120^\circ$  apart. When a single-phase supply is used, it is necessary to split the power supply into two separate coil groups. A capacitance is usually inserted in series with one of the groups to obtain the required phase difference. Other methods can be used, but this is the most common. This causes the single-phase or "split-phase" motor to have characteristics similar in many respects to the polyphase motor.



Note in Figure 2-10 that the sine waves of current traversed  $300^\circ$  through the six positions shown. Accordingly, the field rotated  $300^\circ$ . If the supplied current were 60 Hz, the field would rotate at 60 revolutions per second or 3600 revolutions per minute,  $60 \times 60 = 3600$ . However, if the number of stator coils were doubled, producing a 4-pole field, the field will rotate only half as fast. Thus, the speed of the revolving field is directly proportional to the frequency of the applied voltage, and inversely proportional to the number of stator coils. Perhaps you can see this more clearly in the formula:

$$S = 120f/P$$

where S = speed of rotation of the field (per minute)  
f = frequency of applied voltage (in Hz)  
P = number of poles produced by the 3-phase windings

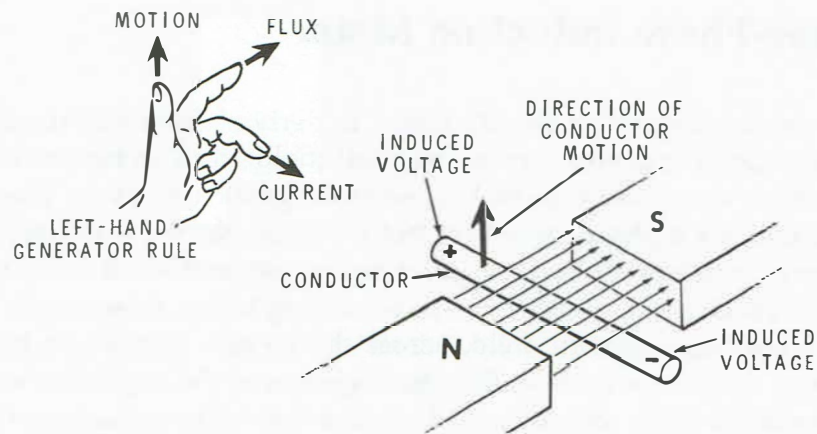
The speed at which an induction motor field rotates is referred to as its “synchronous” speed, because it is synchronized to the frequency of the power supply at all times. A motor with a 2-pole, 3-phase stator winding connected to a 60 Hz source has a synchronous speed of 3600 rpm. A 2-pole, 3-phase, 25 Hz motor has a synchronous speed of 1500 rpm. As we stated previously, increasing the number of stator poles decreases the synchronous speed of the motor. Thus, a 4-pole, 3-phase, 25 Hz motor has a synchronous speed of 750 rpm. A 12-pole, 3-phase, 60 Hz motor rotates at a synchronous speed of 600 rpm. Also, increasing the frequency of the line supply increases the speed with which the motor stator field revolves. Thus, if the frequency is increased from 50 to 60 Hz, and the 3-phase has four poles, the synchronous speed of the field is increased from 1500 rpm to 1800 rpm.

The speed of the rotating field is always independent of load changes on the motor, provided the line frequency remains constant. The magnetic revolving field always runs at the same speed, pole for pole, as the AC generator supplying it. For example, if a 4-pole 60 Hz AC generator runs at 1800 rpm and supplies a 4-pole 60 Hz motor, the motor has a synchronous speed of 1800 rpm. If the same AC generator also supplied an 8-pole 60 Hz motor, that motor would have a synchronous speed of 900 rpm.

## Three-Phase Induction Motor

The driving torque of an AC motor is derived from the reaction of current-carrying conductors in a magnetic field. In AC induction motors, the rotor currents are supplied by electromagnetic induction. The stator windings of a 3-phase induction motor contain three out-of-time/phase currents, which produce corresponding magnetomotive forces, or mmf's. As you recall from our discussion on rotating fields, these mmf's establish a rotating magnetic field, across the airgap, whose synchronous speed is directly proportional to the frequency of the applied power and independent of the motor load. The motor derives its name from the fact that mutual induction, or transformer action, takes place between the stator and the rotor under operating conditions. The magnetic revolving field produced by the stator cuts across the rotor conductors, inducing a voltage in the conductors. This induced voltage causes rotor current to flow. Hence, motor torque is developed by the interaction of the rotor current and the revolving magnetic field. The rotor is not connected electrically to the power source.

Before proceeding with our discussion, let's take a moment and review Fleming's rule for generators and motors. This rule is illustrated in Figure 2-12.



THE LEFT-HAND RULE FOR GENERATORS STATES: If you hold the thumb, first and middle fingers of the left hand at right angles to one another with the first finger pointing in the flux direction, and the thumb pointing in the direction of motion of the conductor, the middle finger will point in the direction of the induced emf. "Direction of induced emf" means the direction in which current will flow as a result of this induced emf. You can restate the last part of the left-hand rule by saying the tip and base of the middle finger correspond to the minus and plus terminals, respectively, of the induced emf.

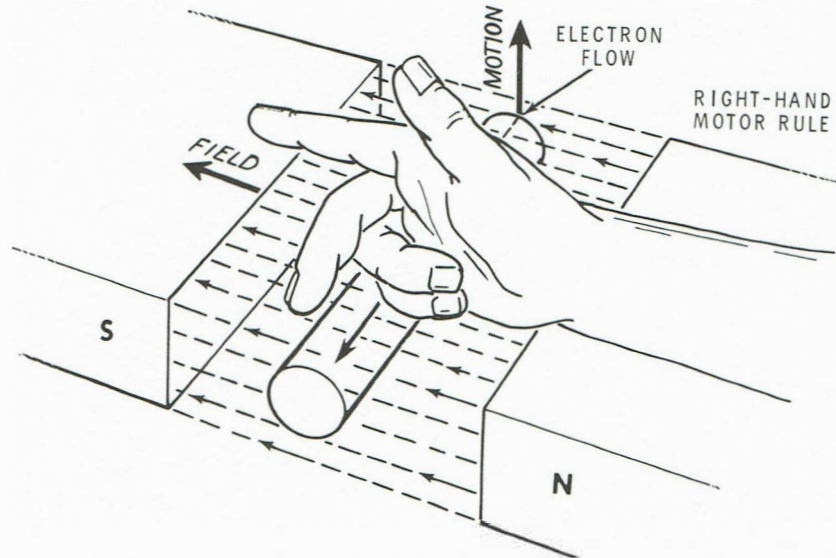


Figure 2-12  
Fleming's Rule  
For Generators and Motors

THE RIGHT-HAND MOTOR RULE STATES: To find the direction of motion of a conductor, the thumb, first finger, and second finger of the right hand are extended at right angles to each other, as shown. The first finger is pointed in the direction of the flux (toward the south pole) and the second finger is pointed in the direction of electron flow in the conductor. The thumb then points in the direction of motion of the conductor with respect to the field. The conductor, the field, and the force are mutually perpendicular to each other.

Figure 2-11 represents the windings of a 3-phase induction motor stator; while Figure 2-13 shows the essential parts — the stator and rotor. The purpose of the iron rotor core is to reduce airgap reluctance and to concentrate the magnetic flux through the rotor conductors. Induced current flows in one direction in half of the rotor conductors, and in the opposite direction in the remainder. The shorting rings on the ends of the rotor complete the path for rotor current.

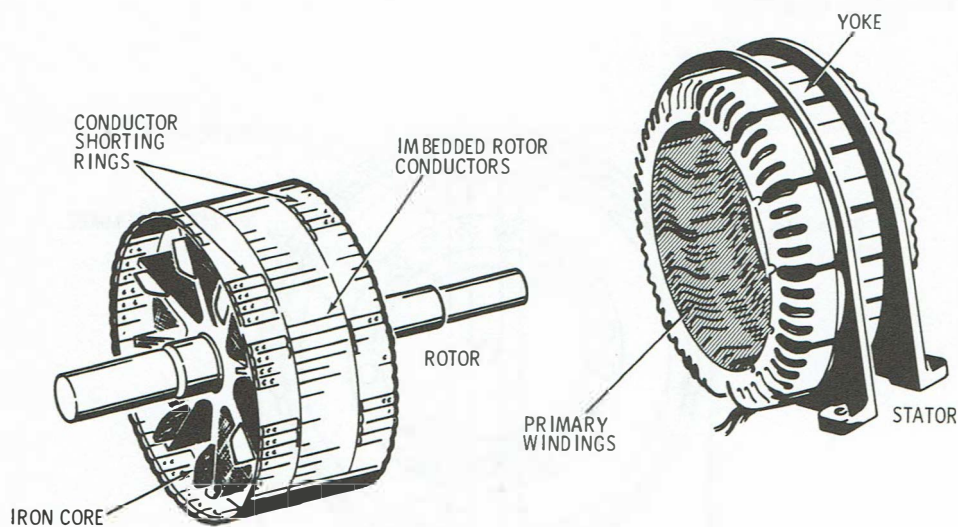


Figure 2-13  
3-Phase Induction Motor Rotor and Stator

In Figure 2-14, a 2-pole field is assumed to be rotating in a counterclockwise direction at synchronous speed. At the instant pictured, the south pole field cuts across the upper rotor conductors from right to left and the lines of force extend upward. Applying the left-hand rule for generator action to determine the direction of the voltage induced in the rotor conductors, the thumb is pointed in the direction of the motion of the conductors with respect to the field. Since the field sweeps across the conductors from right to left, their relative motion with respect to the field is to the right; hence, the thumb points to the right. The index finger points upward and the second finger points into the page, indicating that the rotationally induced voltage in the upper rotor conductors is away from the observer.

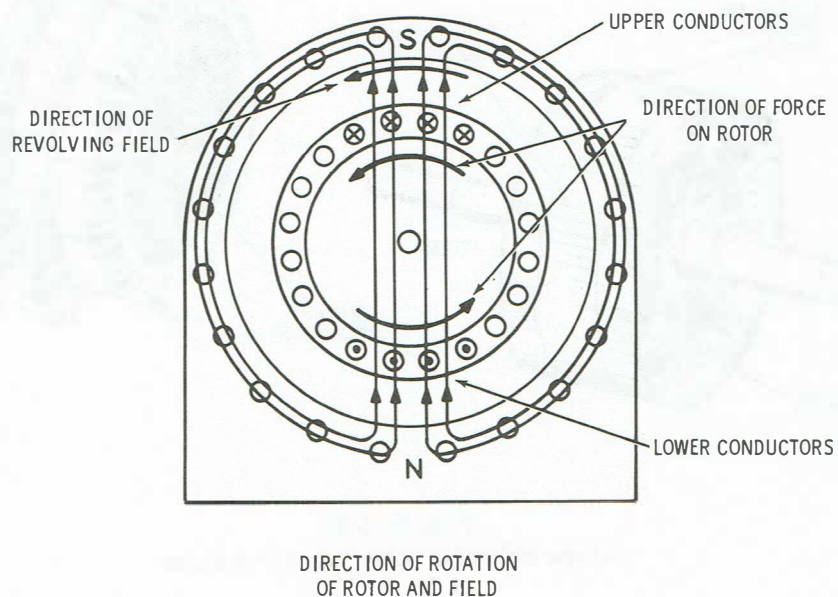


Figure 2-14  
2-Pole 3-Phase Induction Motor



Moreover, applying the left-hand rule to the lower rotor conductors and the north-pole field, the thumb points to the left, the index finger points upward, and the second finger points towards the observer, indicating that the direction of the rotationally induced voltage is out of the page. The rotor bars, or conductors, are connected to end rings that complete their circuit, and the rotationally induced voltages act in series addition to cause rotor currents to flow in the rotor conductors in the direction indicated. For simplification, the rotor currents are assumed to be in phase with the rotor voltage.

You can analyze motor action by applying the right-hand rule for motors to the rotor conductors shown in Figure 2-14, to determine the direction of the force acting on the rotor conductors. For the upper rotor conductors, the index finger points upward, the second finger points into the page, and the thumb points to the left; indicating that the force on the rotor tends to turn the rotor counterclockwise. This direction is the same as that of the rotating field. For the lower rotor conductors, the index finger points upward, the second finger points toward the observer, and the thumb points toward the right; again, indicating that the force tends to turn the rotor in a counterclockwise direction — the same direction as that of the field.

## INDUCTION MOTOR STATOR

The stator of a 3-phase induction motor consists of a laminated steel ring with slots on the inside circumference. The motor stator winding is similar to the AC generator stator winding. Stator phase windings are symmetrically placed on the stator, and like the stator windings of an AC generator, may be either wye or delta connected.

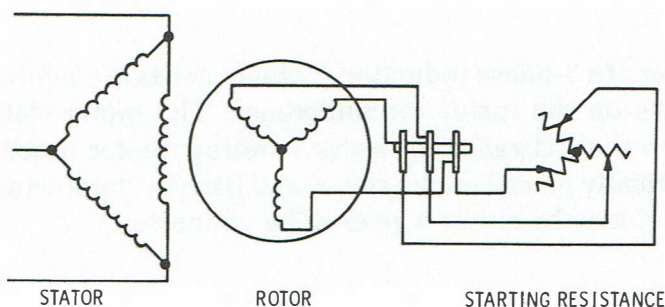
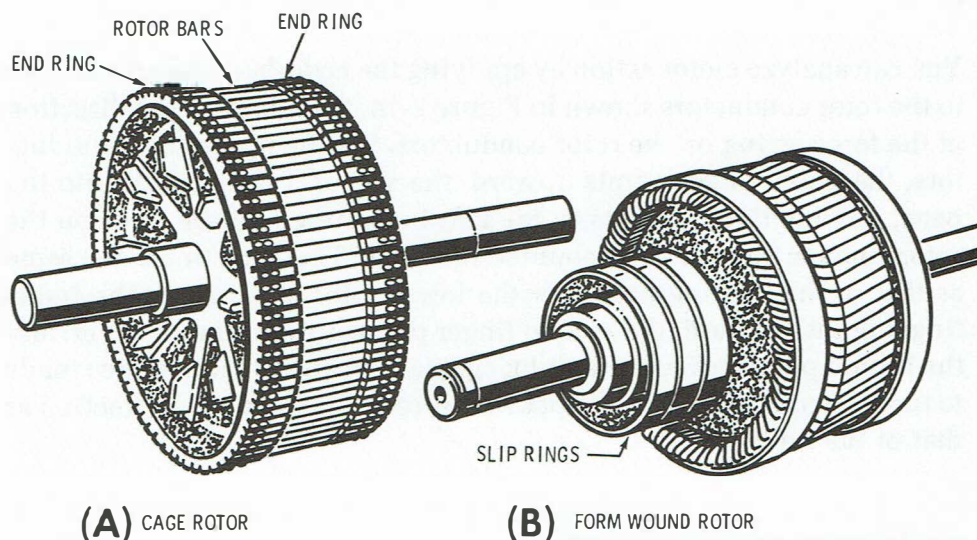
## INDUCTION MOTOR ROTOR

There are two types of rotors — the cage rotor and the form-wound rotor. Both types have a laminated cylindrical core with parallel slots in the outside circumference to hold the windings in place. The cage rotor has an uninsulated bar winding; whereas the form-wound rotor has a two-layer distributed winding with preformed coils.



## CAGE ROTOR

The cage rotor, shown in Figure 2-15A, has rotor bars made of copper, aluminum, or a suitable alloy placed in the slots of the rotor core. These bars are connected together at each end by rings of similar material. The conductor bars carry relatively large currents at low voltages. Hence, it is not necessary to insulate these bars from the core because the currents follow the path of least resistance and are confined to the cage winding.



(C) EXTERNAL VARIABLE RESISTANCE FOR FORM-WOUND ROTOR

Figure 2-15  
3-Phase Induction Motor Rotors

## FORM-WOUND ROTOR

A form-wound rotor, shown in Figure 2-15B, has a winding similar to 3-phase stator windings. Rotor windings are usually wye connected with the free ends of the winding connected to three slip rings mounted on the rotor shaft. An external variable wye-connected resistance, shown in Figure 2-15C, is connected to the rotor circuit through the slip rings. The variable resistance provides a means of increasing the rotor-circuit resis-

tance during the starting period to produce a high starting torque. As the motor accelerates, the variable resistance is cut out. When the motor reaches full speed, the slip rings are short-circuited and the operation is similar to that of the cage rotor.

Regardless of the type of rotor used, the basic principles of operation are the same. The rotating magnetic field generated in the stator induces a magnetic field in the rotor. The two fields interact and cause the rotor to turn. For this reason, the air gap between the rotor and the stator is very small to obtain maximum field strength.

### INDUCTION MOTOR SLIP

As we said previously, the revolving field produced by the stator windings cuts the rotor conductors and induces voltages in the conductors. Rotor currents flow because the rotor end-rings provide a complete circuit. The resulting torque tends to turn the rotor in the direction of the rotating field. If the motor is not driving a load, it will accelerate to nearly the same speed as the revolving field. During the starting period, the increase in rotor speed is accompanied by a decrease in induced rotor voltage because the relative motion between the rotating field and the rotor conductors is less. If it were possible for the rotor to attain synchronous speed, with respect to the stator, there would be no relative motion between the rotor and the rotating field. Therefore, there would then be no induced emf in the rotor, no rotor current, and thus no torque.

It is obvious that an induction motor cannot run at exactly synchronous speed. Instead, the rotor always runs just enough below synchronous speed at no load to establish sufficient rotor current to produce a torque equal to the resisting torque that is caused by the rotor losses. The percentage difference between the speed of the rotating field and the rotor speed is called "slip."

It becomes apparent then that the speed of the rotor, and ultimately the speed of the motor, depends upon the torque requirements of the load. The bigger the load, the stronger the turning force needed to rotate the rotor. The turning force can increase only if the rotor induced emf increases and this emf can increase only if the magnetic field cuts through the rotor at a faster rate. To increase the relative speed between the field and the rotor, the rotor must slow down. Therefore, for heavier loads, the induction motor will turn slower than for lighter loads. Actually, only a slight change in speed is necessary to produce the usual current changes required for normal changes in load. This is because the rotor windings have a very low resistance. As a result, induction motors are often referred to as "constant speed motors."

## Synchronous Motor

With the exception of certain modifications to make its operation more efficient and also to make it self-starting, the synchronous motor is very similar to the rotating field AC generator previously discussed. The rotor fields of both are separately excited from a DC source, and both run at synchronous speeds under varying load conditions. An AC synchronous motor is shown in Figure 2-16.

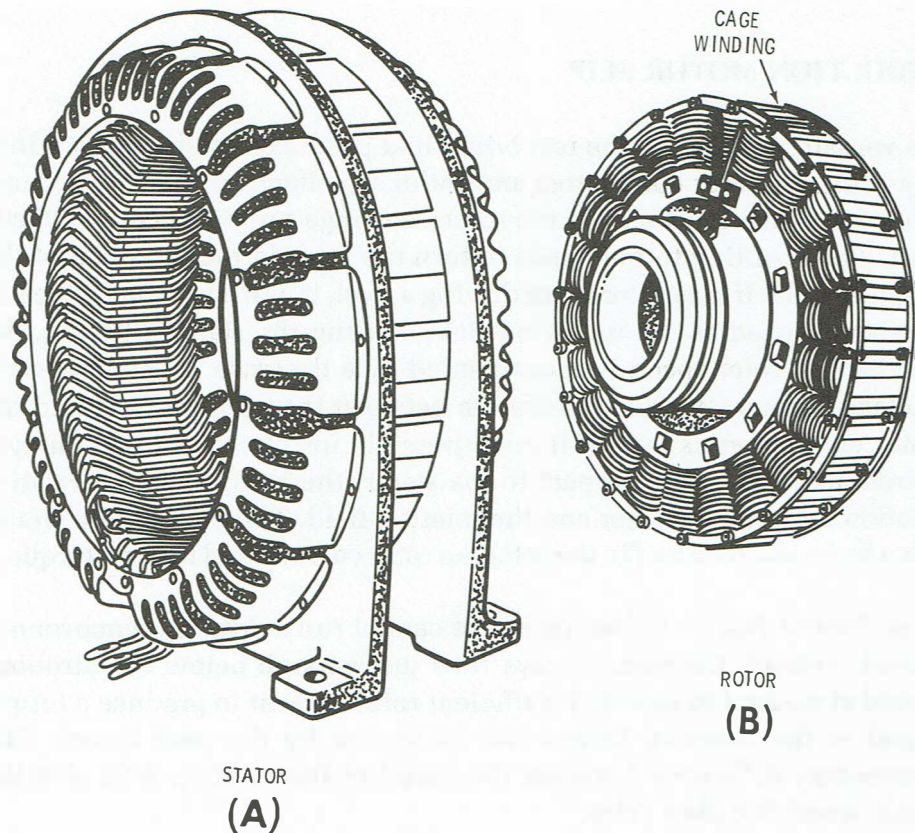


Figure 2-16  
3-Phase Synchronous Motor



## PRINCIPLE OF OPERATION

A polyphase current that is supplied to the stator winding of a synchronous motor produces a rotating magnetic field the same as in the induction motor. A direct current is supplied to the rotor winding, thus producing a fixed polarity at each pole. If it could be assumed that the rotor had no inertia and that no load of any kind were applied, then the rotor would revolve in step with the rotating stator field as soon as power was applied to both of the windings. This, however, is not the case. The rotor has inertia, and in addition there is a load.

The reason a synchronous motor has to be brought up to speed by special means may be understood by observing Figure 2-17.

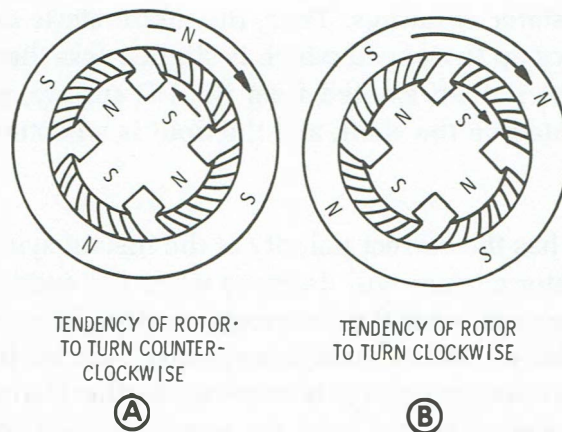


Figure 2-17  
Operating Principles of a 3-Phase  
Synchronous Motor

If the stator and rotor windings are energized, then as the poles of the rotating magnetic field approach rotor poles of opposite polarity, shown in Figure 2-17A, the attracting force tends to turn the rotor in the direction opposite to that of the rotating field. As the rotor starts in this direction, the rotating-field poles are leaving the rotor poles, shown in Figure 2-17B, and this tends to pull the rotor poles in the same direction as the rotating field. Thus, the rotating field tends to pull the rotor poles first in one direction and then in the other; consequently, that characteristic results in zero starting torque.

## STARTING A SYNCHRONOUS MOTOR

As has been explained, some type of starter must be used with the synchronous motor to bring the rotor up to synchronous speed. Although a small induction motor may be used to bring the rotor up to speed, this is not generally done. Sometimes, if direct current is available, a DC motor coupled to the rotor shaft may be used to bring the rotor up to synchronous speed. Once synchronous speed has been attained, the DC motor is converted to operate as a generator to supply the necessary direct current to the rotor of the synchronous motor.

In general, however, another method is used to start the synchronous motor. A cage-rotor winding is placed on the rotor of the synchronous motor to make it self-starting, the same as the induction motor. At start, the DC rotor field is deenergized and a reduced polyphase voltage is applied to the stator windings. Thus, the motor starts as an induction motor and comes up to a speed which is slightly less than synchronous speed. The rotor is then excited from the DC supply, generally a DC generator mounted on the shaft, and the field is adjusted for minimum line current.

If the armature has the correct polarity at the instant synchronization is reached, the stator current will decrease when the excitation voltage is applied. If the armature has the incorrect polarity, the stator current will increase when the excitation voltage is applied. This is a transient condition, and if the excitation voltage is increased further the motor will slip a pole and then come into step with the revolving field of the stator.

If the rotor's DC field winding is open when the stator is energized, a high AC voltage will be induced in it because the rotating field sweeps through the large number of turns at synchronous speed. It is therefore necessary to connect a resistor of low resistance across the DC field winding during the starting period. During the starting period, the DC field winding is disconnected from its source and the resistor is connected across the field terminals. This permits alternating current to flow in the DC field winding. Because the impedance is high, compared with the inserted external resistance, the internal voltage drop limits the terminal voltage to a safe value.

The synchronous motor is ideally suited for applications requiring a constant speed from no-load to full-load condition. This is possible since the synchronous motor has the capability of “locking” the rotor in step with the rotating magnetic field, once the rotor has been brought up to near synchronous speed and the DC field energized.

Before leaving our discussion of polyphase AC motors, it should be noted that most polyphase motors are rated at over 5 horsepower. They are thus suited for large scale industrial applications where large loads have to be moved or positioned.

We will now take a brief look at the smaller single-phase AC motor.

## Single-Phase Motors

Single-phase motors, as their name implies, operate on a single-phase power source. These motors are used extensively in commercial and industrial applications requiring less than 5 horsepower. The advantages of using single-phase motors in small sizes are that they are less expensive to manufacture than other types, and they eliminate the need for 3-phase AC power. Single-phase motors are used primarily in the home to operate refrigerators, washing machines, dryers, etc. They are also used in industry for light tasks such as operating grinders, fans, portable tools, and in small machine applications.

A single-phase induction motor with only one stator winding and a cage rotor is like a 3-phase induction motor with a cage rotor, except that the single-phase motor has no magnetic revolving field at start; hence, it has no starting torque. However, if the rotor is brought up to speed by external means, the induced current in the rotor will cooperate with the stator currents to produce a revolving field. This in turn will cause the rotor to continue to run in the direction in which it was started.

Several methods are used to provide the single-phase induction motor with starting torque. These methods identify the motor as split-phase, capacitor, shaded pole, repulsion, and so forth. We will discuss two of the more common types — the split phase and the capacitor motors.



### SPLIT-PHASE MOTOR

The split-phase motor shown in Figure 2-18, has a stator composed of slotted laminations that contain an auxiliary, or starting winding and a running, or main winding. The axes of these two windings are displaced by an angle of 90 electrical degrees. The starting winding has fewer turns and smaller wire than the running winding; thus, it has higher resistance and less reluctance. The main winding physically occupies the lower half of the slots and the starting winding occupies the upper half. While starting, a centrifugal switch connects the two windings in parallel across the single-phase line that supplies the motor. The motor derives its name from the action of the stator during the starting period.

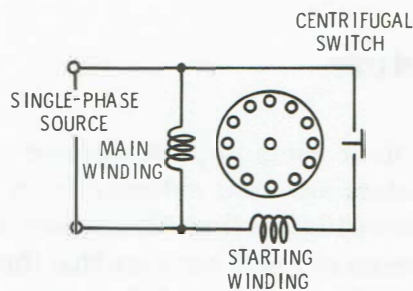


Figure 2-18  
Split-Phase AC Motor

During the starting period, the electrical phase shift of current in the two windings occurs because:

1. The main winding has a high inductance and a low resistance; therefore, current lags voltage by a large angle.
2. The starting winding has a comparatively low inductance and a high resistance; therefore, current lags voltage by a smaller angle.

As an example, suppose the current in the main winding lags the voltage by  $60^\circ$ , and the current in the starting winding lags voltage by only  $30^\circ$ . Hence, the starting currents will therefore be out of phase by  $30^\circ$ ; thus, the magnetic field will be out of phase by the same amount. Although the ideal angular difference is  $90^\circ$  for maximum starting torque, the  $30^\circ$  phase difference will still generate a rotating field of sufficient torque to start the motor.

As the rotating field moves around the air gap, it cuts across the rotor conductors and induces a voltage in them, which is maximum in the area of highest field intensity and therefore is in phase with the stator field. The rotor current lags the rotor voltage at start by an angle that approaches  $90^\circ$  because of the high rotor reactance. The interaction of the rotor currents and the stator field cause the rotor to accelerate in the direction in which the stator field is rotating. During acceleration, the rotor voltage, current, and reactance are reduced and the rotor currents come closer to an in-phase relation with the stator field.

When the rotor has come up to about 75% of synchronous speed, a centrifugally operated switch disconnects the starting winding from the supply line, and the motor continues to run on the main winding alone.

Many of these motors are designed to operate on either 120 volts or 240 volts. For the lower voltage the stator coils are divided into two equal groups and are connected in parallel. For the higher voltage the groups are connected in series. The starting torque of the split-phase motor is 150 to 200 percent of the full-load torque and the starting current is 6 to 8 times the full-load current. The direction of rotation of a split-phase motor can be reversed by interchanging the starting winding leads.

## CAPACITOR MOTOR

The capacitor motor is a modified form of the split-phase motor, having a capacitor in series with the starting winding. A schematic representation of a capacitor motor is shown in Figure 2-19. The capacitor produces a greater phase displacement of currents in the starting and running windings than is produced in the split-phase motor. The starting winding is

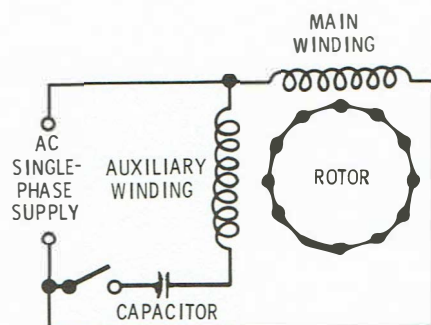


Figure 2-19  
Single-Phase Capacitor-Start AC Motor

made of many more turns of larger wire and is connected in series with the capacitor. The starting winding current is displaced approximately  $90^\circ$  from the main winding current. Since the axes of the two windings are also displaced by an angle of  $90^\circ$ , these conditions produce a higher starting torque than that of the split-phase motor. The starting torque of the capacitor motor may be as much as 350 percent of the full-load torque.

If the starting winding is cut out after the motor has increased in speed, the motor is called a CAPACITOR-START MOTOR. If the starting winding and capacitor are left in the circuit continuously, the motor is called a CAPACITOR-RUN MOTOR. Electrolytic capacitors for the capacitor-start motors vary in size from about 80 microfarads for 1/8 horsepower motors to 400 microfarads for one-horsepower motors. Capacitor motors of both types are made in sizes ranging from small fractional horsepower up to about 10 horsepower. Like the split-phase motor, the direction of rotation of the capacitor motor may be reversed by interchanging the starting winding leads.

This concludes the section on AC motors. After completing the Programmed Review, we will discuss how AC motors are used to provide power to a basic hydraulic and pneumatic unit.

## Programmed Review

- |     |  |
|-----|--|
| 15. | Generally speaking, AC motors are _____ expensive than DC motors.<br>(more/less)   |
| 16. | (less) Before an energized rotor in an AC motor will turn, a _____ magnetic field must be established in the stator windings.  |
| 17. | (rotating) In order to produce a rotating magnetic stator field in a single-phase AC motor, a _____ is usually connected in series with one of the groups of windings. |
| 18. | (capacitor) The speed at which an AC induction motor stator field rotates is referred to as its _____ speed.   |
| 19. | (synchronous) The synchronous speed of an AC induction motor is directly related to the speed of the _____ supplying it.   |
| 20. | (AC generator) The rotor of a three-phase induction motor _____ electrically connected to the power source.<br>(is/is not)   |
| 21. | (is not) The purpose of the iron rotor core, in a three-phase induction motor, is to _____ airgap reluctance.<br>(increase/decrease)                                   |
| 22. | (decrease) The uninsulated conductor bars of a cage rotor carry _____ currents at _____ voltages.<br>(high/low) (high/low)   |
| 23. | (high/low) The speed of rotation of an AC induction motor depends upon the _____ requirement of the load.  |
| 24. | (torque) In an AC synchronous motor, a _____ current is supplied to the rotor winding.   |

25. (direct) An AC synchronous motor starts as an \_\_\_\_\_ motor.

26. (induction) When the split-phase induction motor has reached approximately 75% of its synchronous speed, a \_\_\_\_\_ operated switch disconnects the starting winding from the supply.

27. (centrifugally) The starting torque of a split-phase motor is \_\_\_\_\_ the starting torque of a capacitor motor.  
(less than/greater than)

(less than)

## BASIC HYDRAULIC SYSTEM

Now that we have studied AC power generation and one of its main applications — supplying AC motors, let us now put these AC motors to work powering a hydraulic system. Hydraulic systems are used in many industrial applications and robotics is no exception. Many of today's industrial robots, especially the larger ones, incorporate hydraulics in one form or another. A robot's hydraulic system can be either self-contained, within the robot's base or pedestal, or externally mounted nearby. Whichever is the case, they perform the same function — providing fluid power for robot positioning.

The basic hydraulic system, shown in Figure 2-20, can be compared to a simple electrical circuit. The oil reservoir (1) stores the hydraulic fluid used in the system. In this capacity, it acts much like an AC generator in

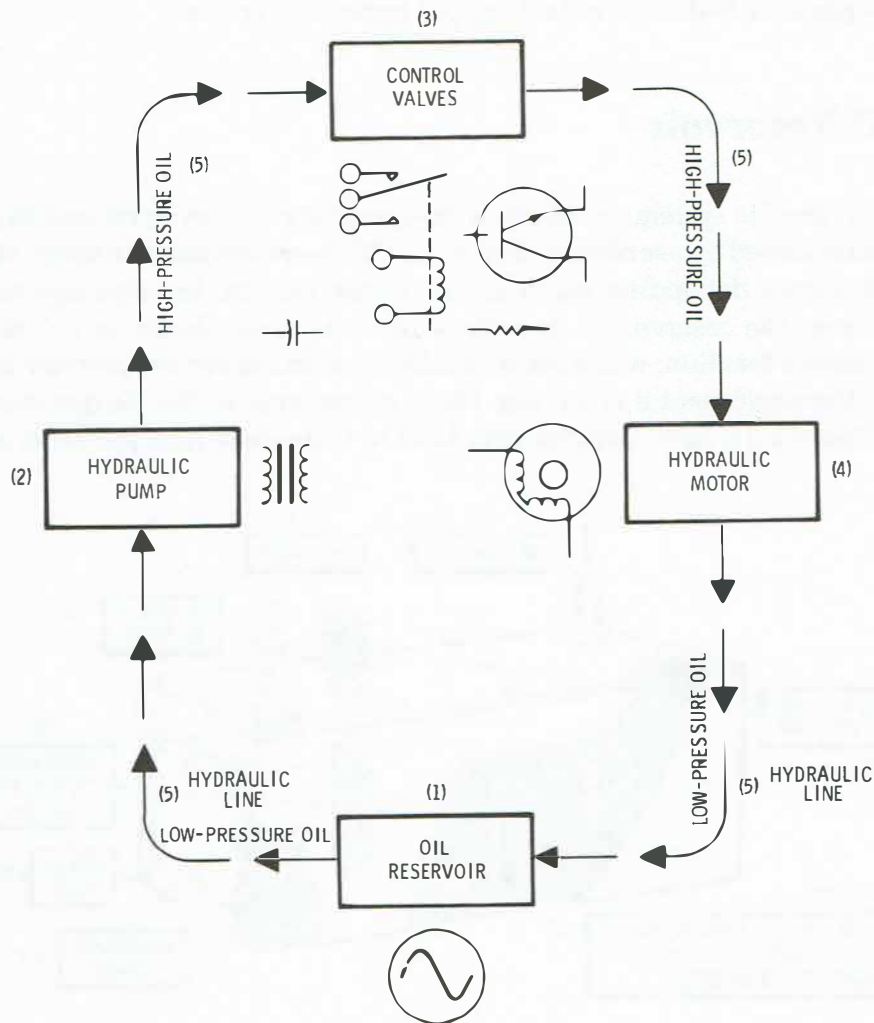


Figure 2-20  
Basic Hydraulic System



that there is a constant source of power for the system. The hydraulic pump (2) draws the hydraulic fluid out of the reservoir, and once it has increased the pressure, similar to a transformer increasing voltage in a circuit, it distributes the high-pressure hydraulic fluid to the rest of the system. The hydraulic control valves (3) can be compared to electronic control components, such as resistors, capacitors, transistors, and relays, in that they are used to control the rate, amount, and direction of hydraulic fluid flow. The hydraulic motor (4) converts the high-pressure hydraulic fluid into rotary or linear motion, through the use of rotary actuators or cylinders and pistons, to perform some useful task. In this capacity, the hydraulic motor can be compared to an electric motor in providing rotary motion or to the solenoid which can provide linear motion. The hydraulic oil lines or pipes (5) can be compared to electrical conductors; the hydraulic lines carry the hydraulic fluid while electrical conductors provide a path for current to flow. This has been a brief overview of a hydraulic system. We will now take a closer look at the components that constitute a simple hydraulic system.

## Oil Reservoir

All hydraulic systems must have the capability of storing oil and this is accomplished by use of an oil reservoir. Oil reservoirs come in many sizes and shapes depending on their use in the system. In large hydraulic systems, the reservoir is usually a separate tank placed in an easily accessible location; while many smaller systems have the reservoir built into the equipment it is serving. Many oil reservoirs, like the one shown in Figure 2-21, have extra features built in to do more than just store oil.

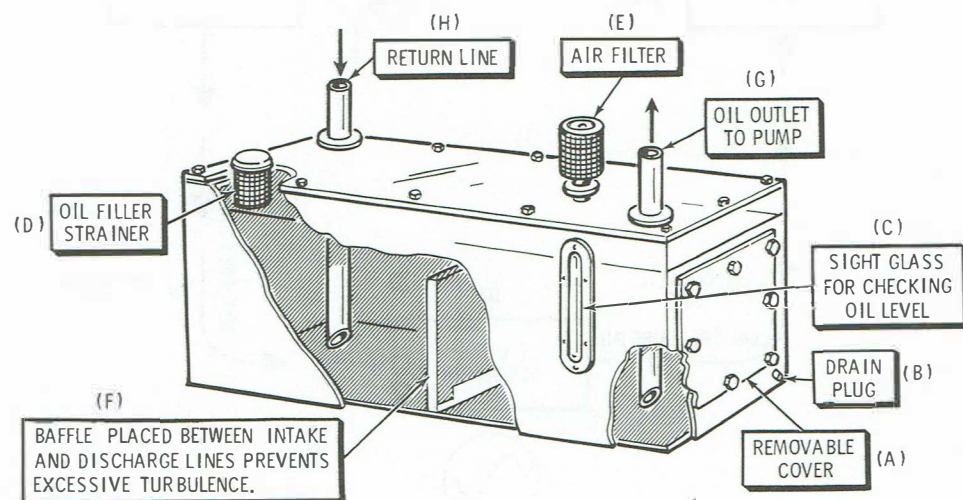


Figure 2-21  
Hydraulic System Oil Reservoir

The reservoir is usually constructed of welded steel plate with removable covers (A) at each end to permit easy access for cleaning. The bottom of the tank is rounded, with a drain plug (B) located at the lowest point, to allow for complete draining of the tank. The reservoir usually contains either a sight-glass (C) or a dip-stick for checking fluid level. The reservoir is filled through a filler tube (D) that uses a fine wire mesh screen to prevent contaminants from entering the tank during fluid replenishment.

If the reservoir is not pressurized a filtered air inlet is required. The air filter (E) must be large enough so as to maintain atmospheric pressure in the tank regardless of fluid level. The air filter can be of many different designs, but when used in a dirty environment an oil bath type is generally used. If a pressurized system is being used, an air filter is not required and is replaced by an air valve to regulate pressure.

The baffle plate (F) is placed inside the reservoir in such a way as to isolate the pump outlet line from the return line. The baffle plate helps reduce turbulence inside the tank by not allowing the same fluid to circulate continuously, but take a different path through the tank. This baffle plate also helps get rid of trapped air in the system, and it acts as a barricade to contaminants by allowing them to settle to the bottom of the tank.

The pump outlet line (G) and the system return line (H) must be placed well below the fluid level in the tank. If they are terminated above the fluid level this can cause air to be drawn into the system. Lines that terminate near the bottom of the tank should be cut at a 45 degree angle. This prevents the line opening from "bottoming" in the tank and cutting off oil flow. It is even more desirable if these lines can terminate in a strainer or filter. All lines should be tightly sealed to prevent air or dirt from entering the system.

The reservoir should be of sufficient size to store all the fluid in the system, and also allow for heat expansion of the fluid. It is a generally accepted rule that the reservoir should hold two or three gallons of fluid for each gallon per minute of pump delivery.

## Hydraulic Pumps

The pump could be referred to as the heart of the hydraulic system, as it is probably the most important component in the system. It is easy to think of the hydraulic pump as a sort of compressor, but this is not so. Oil is virtually incompressible, at low pressures; even though it can be compressed somewhat at very high pressures. Try not to think of the pump as a compressor, because the purpose of the pump is not to compress, but to create a force.

### VANE PUMP

The vane type of pump, shown in Figure 2-22 is used extensively to produce hydraulic power. It produces a relatively steady flow and is capable of creating pressures as high as 2,000 psi (pounds per square inch). Some vane pumps, under low pressure, have a discharge capacity of 35,000 gallons per minute. Wear does not greatly decrease efficiency, because the vanes can always maintain a close contact with the ring within which they rotate. The close fit of the rings may be destroyed if dirt and sludge in the oil causes them to stick in their slots.

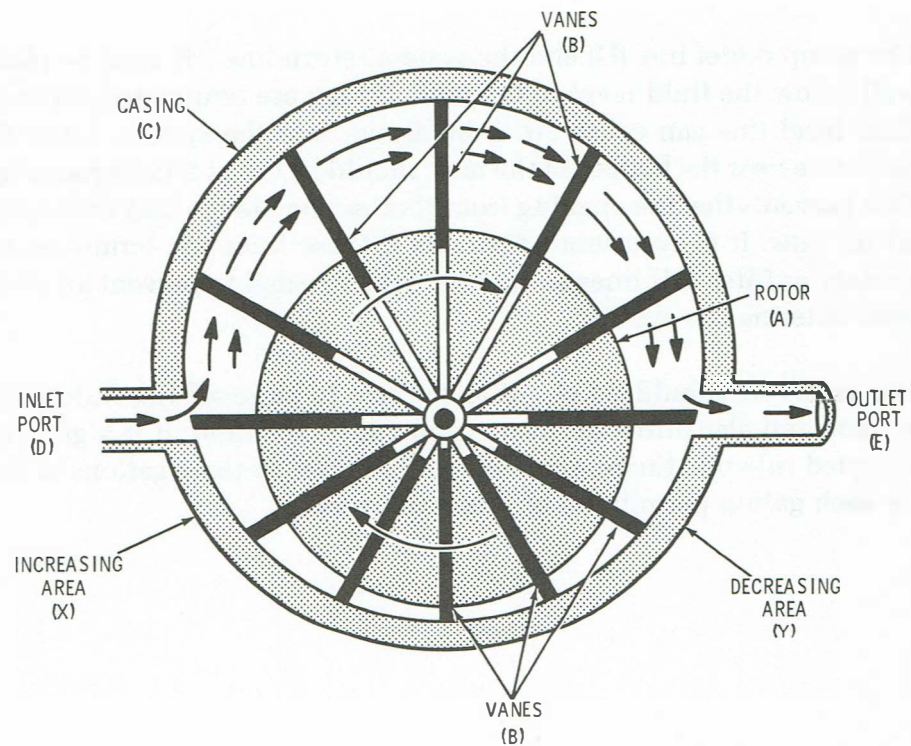


Figure 2-22  
Rotary Vane Pump



The operation of the vane pump uses the principle of moving the oil around the pump casing to create a pressure. The vane pump has a rotating member (rotor) (A) eccentrically located with regard to the casing and is provided with a number of spring-loaded radial vanes (B) which ride on the casing (C). When the rotor is turned clockwise, by an external force, the area between the vanes and the casing increases at point (X) and allows oil to be drawn into the pump through the inlet port (D). This oil is carried, in the area between the vanes and the casing, toward the outlet port (E). Once the oil has reached this area, the area between the vanes and the casing has decreased, at point (Y), thereby forcing the oil out of the pump under pressure.

In the rotary pump just described, discharge can be varied only by changing the speed of the driving motor. With AC motors this involves a relatively complex electronic or mechanical mechanism, hence rotary pumps have been developed which incorporate a means of discharge adjustment within the pump. These pumps are known as variable-displacement or proportioning pumps. Most rotary pumps are self-priming and are capable of providing a suction which will lift a fluid to a height of 25 feet.

## Hydraulic Valves

While we referred to the pump as the heart of the hydraulic system, we can refer to the valves as the control portion of the system. A valve can be used for pressure control, directional control, flow control, and in some cases for controlling other valves. Until a few years ago valves were the only means of controlling flow and pressure, but now there are several pumps available with means of varying flow and pressure. Nevertheless, valves are still the most important component for providing flexibility in all complex hydraulic systems. These valves range from very simple to extremely complex in design and function. The valves we will discuss are by no means the only valves associated with a hydraulic system, but have been chosen because they represent the specific types of valves most commonly used.

## DIRECTIONAL CONTROL VALVES

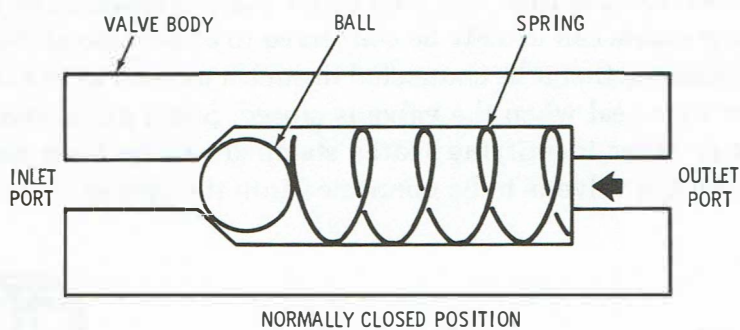
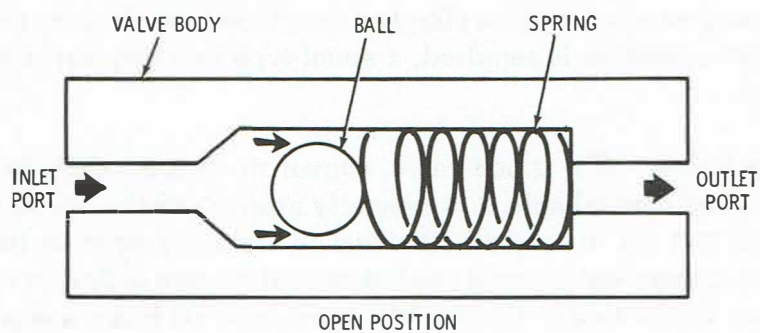
In order for a hydraulic system to perform a desired function it is essential that the direction of fluid flow be controlled. Directional control of the fluid is obtained by using valves especially designed for this purpose. These valves are designed to start, stop, or reverse system flow without causing a perceptible change in system pressure or flow rate. The more common directional control valves are usually referred to by the number of flow paths, i.e., one, two, three, or four-way valves. These valves may be actuated manually, mechanically, electrically, or by pressure.

### One-way Valves

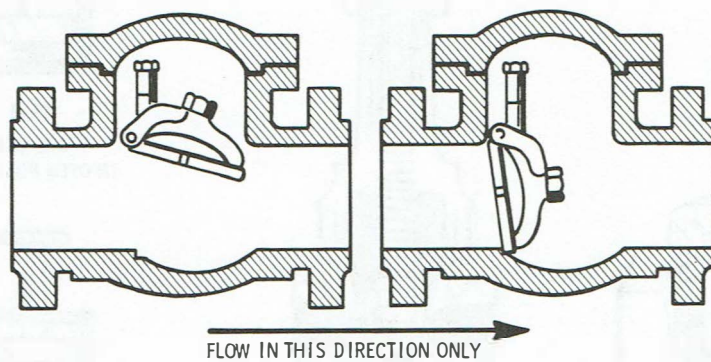
One-way valves, or check valves as they are commonly referred to, permit flow in only one direction.

The ball type check valve, shown in Figure 2-23A, is found in hydraulic lines where backflow cannot be tolerated. A light spring holds the ball so that the valve is normally closed, therefore permitting the valve to be installed in any position. The pressure at which the valve starts to open, in the free flow direction, is called the "cracking pressure". This cracking pressure is usually in the neighborhood of 5 psi, but can be as high as 3000 psi in certain special applications. Valves of this type are also used for special requirements such as bypassing heat exchangers or filtering units in the event of high flow surges or clogging. When used in this manner, they are not being used as check valves, but rather as relief valves.

Like the ball type check valve, the flapper valve, shown in Figure 2-23B, allows flow in only one direction. The flapper is often referred to as a "swing check" valve because of the motion of the flapper, which is hinged on one side and is free to swing like a gate. Unlike ball valves which are available in only small sizes, the construction of the flapper valve makes it usable in comparatively larger sizes. Also, when fully open, it offers very little resistance to flow, which in turn reduces the amount of turbulence created by the valve. Flapper valves are normally mounted vertically with flapper weight, gravity, and fluid pressure holding them closed. When horizontal installation is required they are fitted with a light spring to assist in closing when the fluid has stopped moving.



**(A)** BALL-TYPE CHECK VALVE



**(B)** FLAPPER VALVE

Figure 2-23  
One-Way Directional Control Valves



## Two-way Valves

Two-way valves are often simple shut-off devices. If more than simple shut-off operation is required, a spool-type two-way valve may be required.

**Globe Valve** — The globe valve, shown in Figure 2-24A, is simple in design and very reliable. It is generally used where the pressure does not exceed 150 psi. It works best when in the fully open or fully closed position; therefore, it is not used to control the rate of flow in a hydraulic system. When used in the partially open position leaks may occur along the stem allowing air to enter the system. The globe valve offers some resistance to fluid flow and may cause some turbulence in the system. While pressure can usually be connected to either side of the valve, it is recommended that it be connected in such a manner as to keep pressure off the stem seal when the valve is closed. Some globe valves have an arrow or other identifying marks stamped on the body denoting the direction the valve is to be connected into the system.

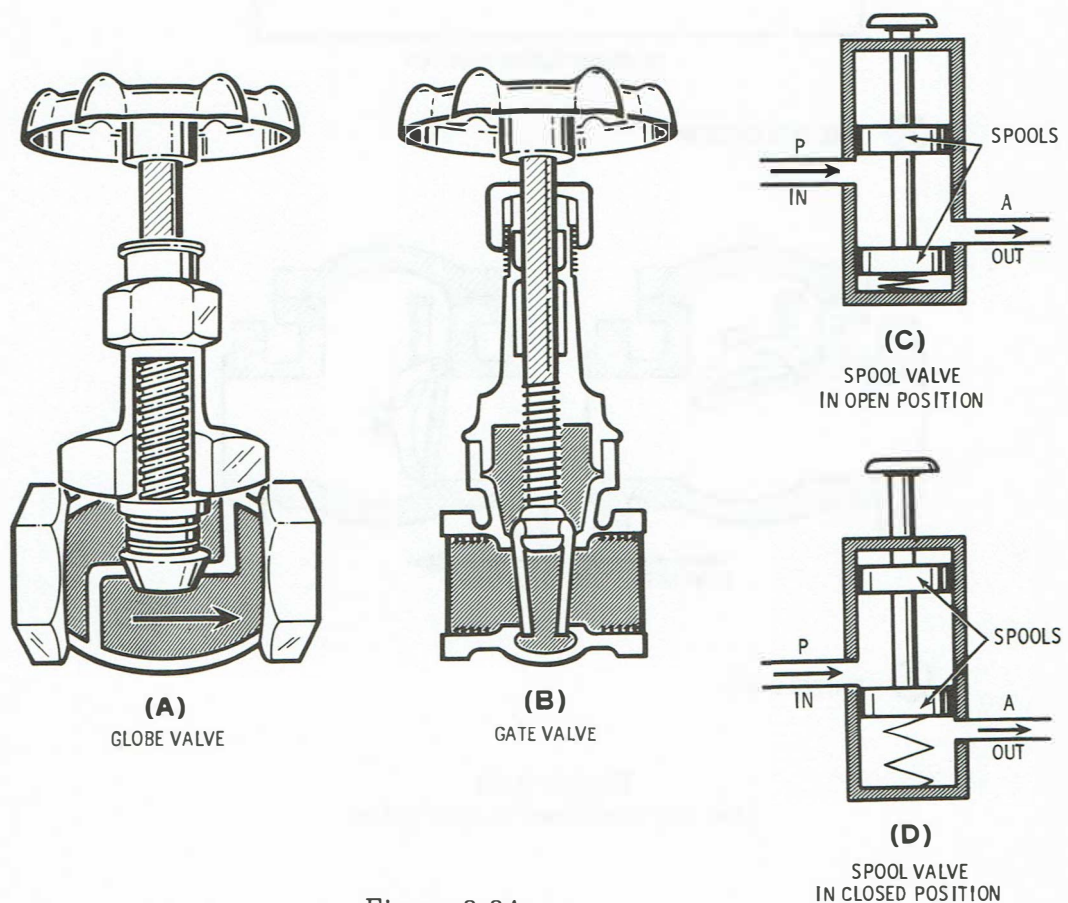


Figure 2-24  
Two-Way Directional Control Valves

**Gate Valve** — The gate valve, shown in Figure 2-24B, is used when higher pressures and greater flow is required. A gate valve can handle pressures up to 5000 psi, and, unlike the globe valve, can handle the pressure in either direction. Like the globe valve it is designed for use either fully open or fully closed. If it is used in the partially open position rapid gate wear will occur, due to the high pressure, and this will in turn cause the valve to leak when in the closed position. One advantage of the gate valve is that when it is fully open it offers virtually no resistance to flow and therefore causes little pressure loss or turbulence in the system. Also, even very large gate valves can be controlled quite easily by automatic mechanisms.

**Spool Valve** — The spool or piston type valve is probably the most common type of directional control valve used in a hydraulic system. It gets its name from the fact that the valving elements look like spools. Figure 2-24C shows a basic two-way spool valve. Note that there are two spools, or pistons, connected to the valve rod. This is done so that the valve can be easily actuated even though the fluid might be at a very high pressure. If only one spool were used the full pressure of the fluid would be applied to it and the spool would require a great deal of force to actuate it. Using two spools, the pressure is applied equally to both spools, but the forces are in opposite directions and the net force is zero.

In some instances, a spool-type valve may be used to partially control the flow of fluid. Actuation of the spool valve may be accomplished manually (by a lever), electrically (using a solenoid), or by mechanical methods such as cam rollers, or foot pedals. A two-way spool valve is said to be normally closed (abbreviated N.C.) when it blocks the flow of fluid through the valve as seen in Figure 2-24D. Consequently, it is said to be normally open (abbreviated N.O.) when fluid is permitted to flow through the valve. This normally open or normally closed condition is in reference to the valve being in its nonactuated position. Valves without return springs do not have a “normal” position.

The precision machining and fitting of the mating surfaces requires that the hydraulic fluid be kept as clean and sludge-free as possible. Contamination in the fluid will cause rapid wear of valve parts, which in turn will cause leakage and inefficient operation.

### Three-Way Valve

Another type of spool valve is the three-way valve shown in Figure 2-25. It can handle certain valving functions beyond the capability of a shut-off or two-way valve. In hydraulics the most important application for a three-way valve is for directional control of a single-acting ram or cylinder. It may also be used to control hydraulic motors. Like the two-way valve, it is classified as normally open or normally closed, when in a non-actuated condition. This non-actuated condition is referred to as the “normal” position. A three-way valve may use a spring force to return to this normal position. The port markings on the valve are assigned by the American National Standard Institute (ANSI). The letter “P” is a designation of pressure; letter “A” represents actuating port; and letter “E” or “T” indicates exhaust or tank return. There are numerous variations of the three-way valve with many different applications used today.

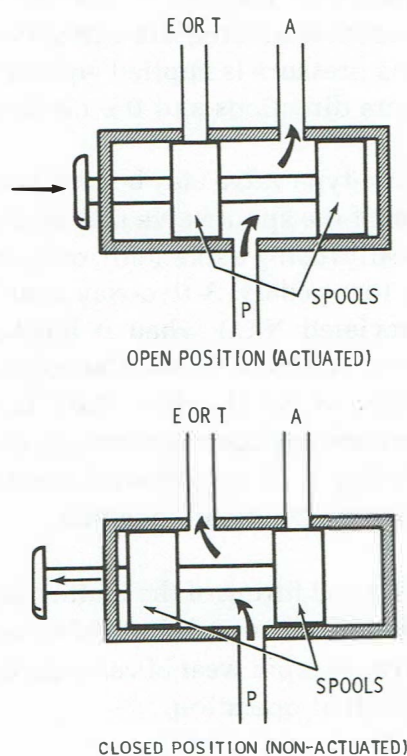


Figure 2-25  
Three-Way Directional Control Spool Valve

### Four-way valves

Four-way valves, because of their many different designs, can be used in almost any control application where starting, stopping, or reversing the direction of flow is required. Four-way valves are used to control the forward and reverse action of double-acting cylinders, and also to reverse rotation of a hydraulic motor. Actuation and the marking of valve ports is basically the same as the two and three-way valves previously discussed. Figure 2-26 shows a basic four-way spool valve in various positions, with all ports marked.

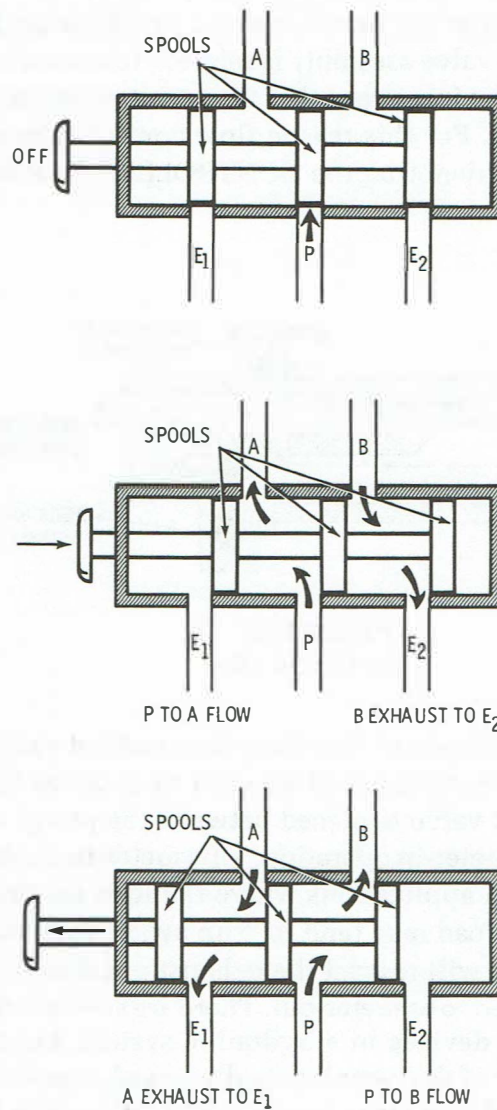


Figure 2-26  
Typical Four-Way Directional Control Valve



## FLOW CONTROL VALVES

The rate at which hydraulic fluid is delivered to the load of a system determines its operational speed. To regulate this speed, flow or volume control devices are used. This flow could be regulated by using a variable displacement pump, but this would limit the capabilities of a system where more than one speed is required.

The most common type of flow control valve used is shown in Figure 2-27. The designations P and F, again ANSI designations, refer to pressure and free flow connections respectively. Flow is controlled in one direction only, in this case from left to right, with the amount of flow regulated by the setting of the needle valve. Fluid flowing from right to left, through the check valve assembly is referred to as moving in the free flow direction. It is very important that flow control valves be properly installed in the system. For this reason flow control valves are usually marked with an arrow denoting the CONTROLLED flow direction.

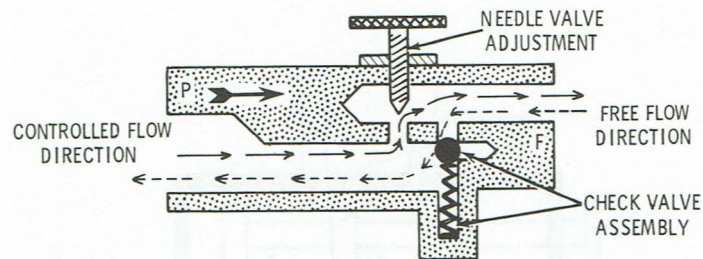


Figure 2-27  
Flow Control Valve

There are two basic methods of installing flow control valves to control load speed. The term metering is often used to describe this function. When the flow control valve is placed between the pump and the load they are referred to as meter-in operation. This meter-in method is highly accurate and is used in applications where the load continually resists movement. Where the load may tend to “run away” flow control valves are located where they will restrict the exhaust or return flow from the cylinder. This is referred to as meter-out. There are several other methods of installing metering devices in a hydraulic system, but they are just different combinations of the two already discussed. Some of these other methods use check and by-pass valves in conjunction with flow control valves.

## PRESSURE CONTROL VALVE

Some hydraulic systems require more than one pressure level during operation. These different pressure levels may also be required at different intervals during the system's operation. Those valves which control the pressure level are classed as pressure control valves. Included in this group are other valves such as reducing, by-pass, sequencing, and pressure relief valves.

Figure 2-28 shows a very basic adjustable relief valve used for pressure control. This type of valve is found in virtually every hydraulic system. It is connected between the pump outlet and the reservoir and is normally closed. Its purpose is to limit the system pressure to some predetermined level by diverting some or all of the pump's output to the reservoir when that pressure level has been reached. This valve is sometimes incorporated into the pump itself.

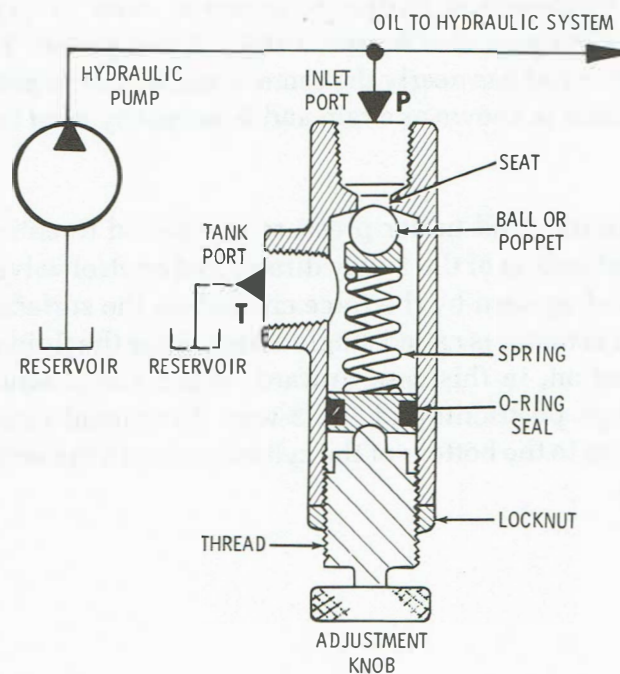


Figure 2-28  
Pressure Control Valve

When the fluid pressure exceeds the setting of the adjustable spring force, due to excessive pump output or system overload, the ball or poppet is forced off its seat, allowing some or all of the fluid to flow to the reservoir. When the excessive pressure is removed, the ball or poppet will return to its seat and normal operation will resume.



## Hydraulic Actuators

Until now, we have developed hydraulic pressure by using a pump; regulated this pressure by using a pressure control valve; controlled the amount of flow required by means of a flow control valve; and even directed where we wanted this pressurized fluid to go, through the use of a directional control valve. It is now time to change this controlled hydraulic power into useful mechanical power. This is accomplished by hydraulic actuators. Hydraulic actuators are known as pistons, cylinders, rams, and even motors. Before leaving this section on hydraulics, we will take a brief look at some of the devices used to convert fluid power into mechanical power.

### SINGLE-ACTING ACTUATOR

The basic single-acting hydraulic actuator, seen in Figure 2-29A, is essentially a cylinder that houses a tight fitting piston. You will notice that the piston rod has nearly the same diameter as the piston itself. This type of actuator is known as a ram and is primarily used for lifting heavy objects.

When hydraulic fluid under pressure is allowed to enter the cylinder, through positioning of the 3-way directional control valve, the piston or ram is moved upward by the force created on the surface of the piston. This type of actuator is called single acting since the fluid moves the ram in one direction, in this case upward. When the pressure is released, again through positioning of the 3-way directional control valve, the piston returns to the bottom of the cylinder, due to the weight of the load on the ram.

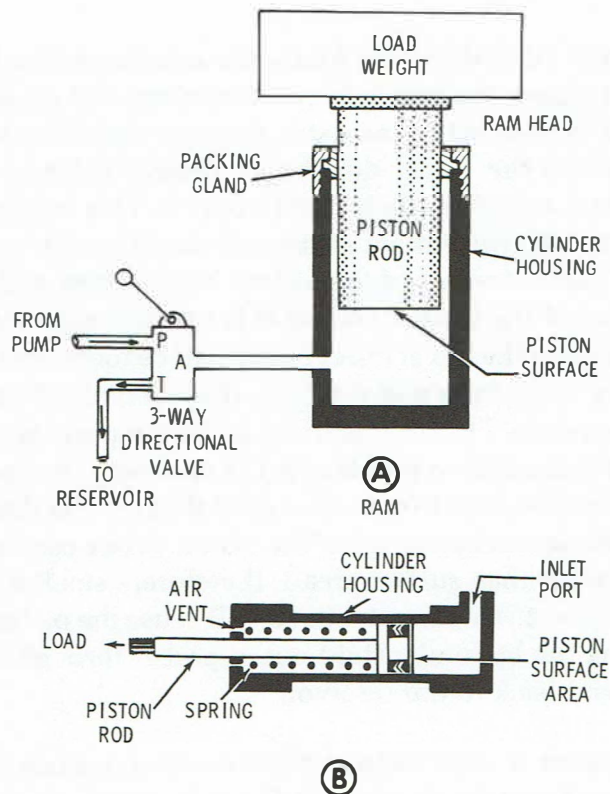


Figure 2-29  
Single-Acting Hydraulic Actuator

Another, smaller type of single-acting actuator is shown in Figure 2-29B. This actuator uses a piston with a much smaller piston rod. When fluid enters the cylinder from the right, it causes the piston to move to the left, due to the force felt on the piston surface area. When the pressure is removed, the piston will return to the right due to the action of the spring. This type of actuator can generally be mounted in any attitude, due to the spring return, and is used where a great force is required in only one direction. Like the larger ram, this smaller actuator is usually controlled by a 3-way directional control valve.

## DOUBLE-ACTING ACTUATOR

Figure 2-30 depicts a double-acting hydraulic actuator where hydraulic pressure is used to move the piston in two directions. Let us observe the operation of this double-acting actuator through one complete cycle. Suppose we position the 4-way directional control valve so that P is connected to port A and T is connected to port B. This will permit the hydraulic fluid to enter the bottom of the cylinder, through port A, and create a force on surface area 1 of the piston. This in turn will cause an upward movement of the piston. During this upward movement of the piston, hydraulic oil in the top of the cylinder will be forced out of port B, through the 4-way valve from port B to T, to the reservoir. Now, supposing the piston has reached its limit of travel and we reposition the 4-way valve so that P is connected to port B and T is connected to port A. This will allow the hydraulic fluid to enter the top of the cylinder through port B, and create a force on surface area 2 of the piston. As we can see, surface area 2 is much smaller than surface area 1; therefore, a smaller force will be felt on surface area 2. This smaller force will cause the piston to move downward forcing the hydraulic fluid out of port A through the 4-way valve, to port T and back to the reservoir.

Although this actuator is referred to as double-acting, we can see that it will provide a greater force in the upward direction, due to the difference in piston surface areas.

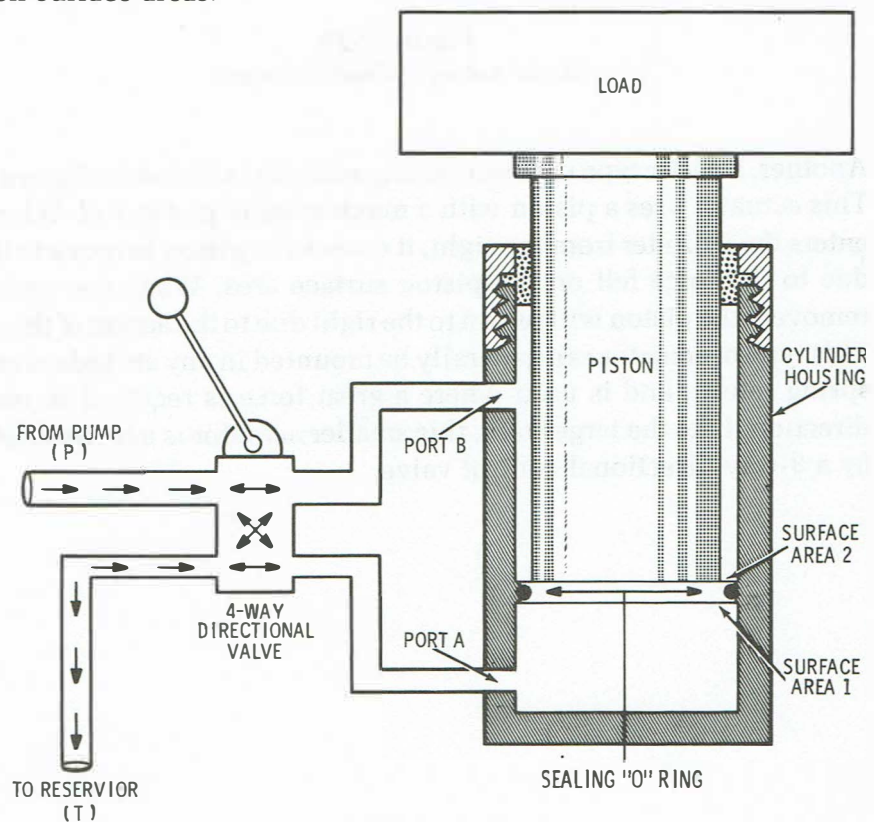


Figure 2-30  
Double-Acting Hydraulic Actuator

Figure 2-31A shows the operation of a small double-acting cylinder. Here again fluid can be applied, by means of a directional control valve, to either surface area of the piston, thereby moving the load in either direction. Again, we see that, due to the difference in surface areas of the piston, more force will be applied to extend the load than to retract it. This type of double-acting cylinder is referred to as a differential, or unbalanced type cylinder, and the piston rod extends from only one end of the cylinder.

The double-acting cylinder shown in Figure 2-31B is referred to as a nondifferential or balanced type of actuator. Here there is a piston rod on each side of the piston, hence the piston surface areas are equal. The force will be the same in either direction as long as the pressure remains the same. While this type of actuator is usually used to produce linear motion, it can be used with certain mechanical devices to provide rotary motion when required.

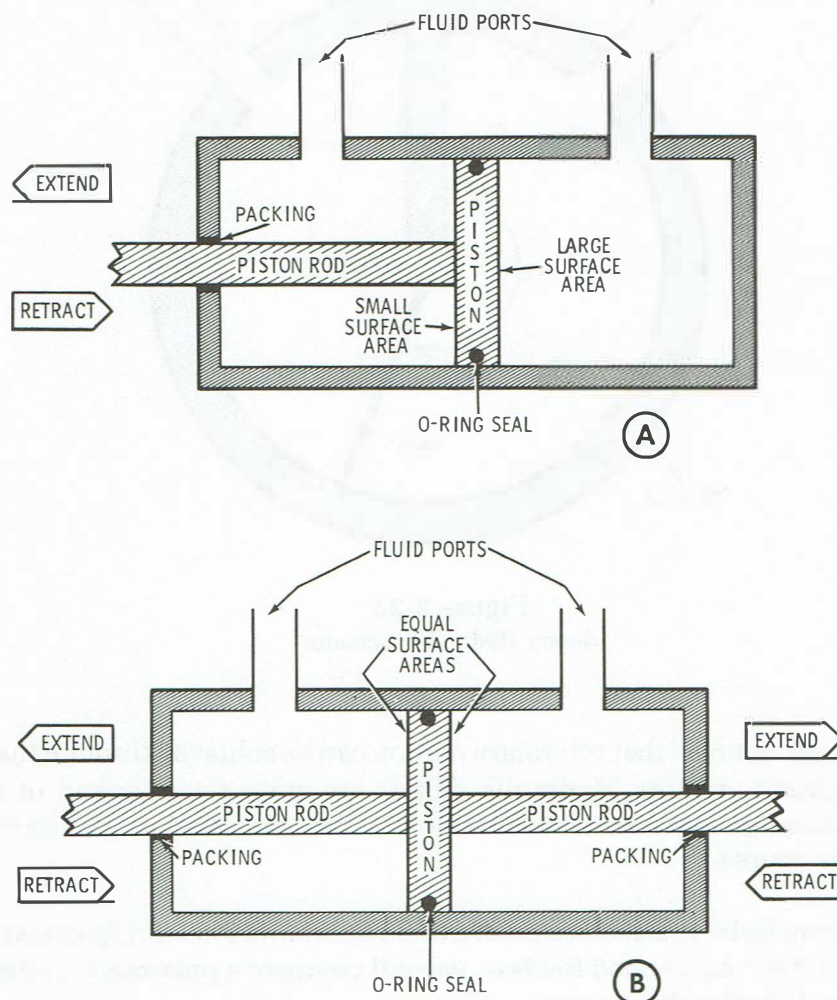


Figure 2-31  
Double-Acting Hydraulic Cylinders



## ROTARY ACTUATOR

Figure 2-32 illustrates a very basic type of hydraulic actuator designed to produce a limited amount of rotary motion. This rotary actuator can be made to turn approximately  $280^\circ$  in either direction. When fluid is applied to port A and expelled from port B, through the action of a directional control valve, the vane will rotate in a counterclockwise direction. When this process is reversed, applying fluid to port B and expelling fluid from port A, clockwise rotation will occur. More turning power can be obtained by using a double-vane rotary actuator, but this extra power is gained at the expense of rotary movement. Double-vane rotary actuators can only be moved approximately  $100^\circ$  in either direction. Rotary actuators are frequently used when continuous reciprocating operations are required.

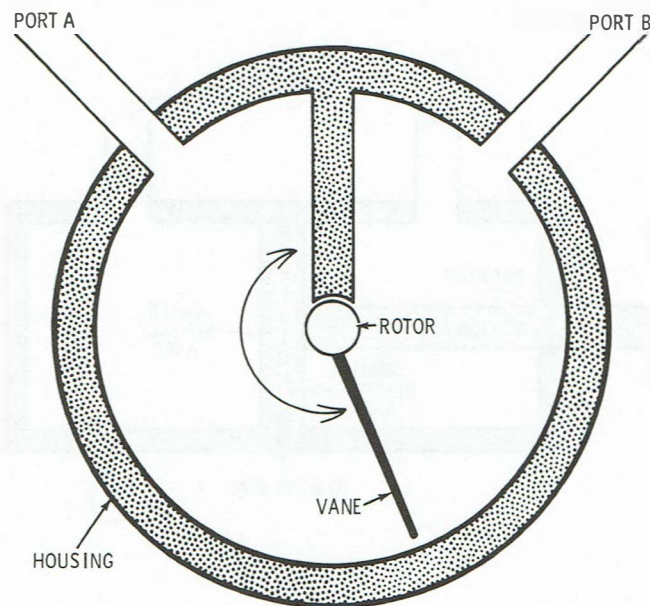


Figure 2-32  
Rotary Hydraulic Actuator

It should be noted that full rotary motion can be achieved through the use of hydraulic motors. Hydraulic motors are quite complex and of such varied design that further discussion of such devices is beyond the scope of this course.

This concludes the discussion of a basic hydraulic system. Upon completion of the Programmed Review, we will compare a pneumatic system to the basic hydraulic system.



## Programmed Review

28.	The _____ stores the oil in a hydraulic system.
29.	(oil reservoir) The _____ type pump is the most common type of pump used to provide hydraulic power.
30.	(vane) A _____ valve is used to start, stop, or reverse system fluid flow.
31.	(directional control) A _____ or check valve permits flow in only one direction.
32.	(one-way) A _____ valve can be easily actuated even if the fluid pressure is very high.
33.	(spool) A _____ valve is used to control flow rate in a hydraulic system.
34.	(flow control) A _____ valve is used to limit system pressure to some predetermined level.
35.	(pressure control) The _____ is the component in a hydraulic system that does the actual work.
36.	(actuator) A _____ is used primarily for lifting heavy objects.
37.	(ram) A _____ actuator can provide a force in two directions.
38.	(double-acting) Full _____ motion can be achieved through the use of hydraulic motors.
	(rotary)

## PNEUMATIC SYSTEMS

A Pneumatic system is very similar, both in construction and operation, to the previously discussed hydraulic system. Whereas the hydraulic system used a non-compressible fluid to transmit a force between the source and the load, a pneumatic system uses compressible air to perform the same function. Before we proceed with our discussion of a basic pneumatic system, let's take a few moments and compare some of the differences between the two systems.

Both, hydraulic and pneumatic systems are very reliable and, with the proper methods of control, are extremely accurate positioning devices. Because of their extreme accuracy, hydraulics and pneumatics are used in a wide variety of industrial robotic applications where accurate placement and repeatability is a must. Hydraulic systems are used when a heavy load is to be positioned or handled; whereupon, pneumatics are used to position relatively light loads. Many industrial applications, robots included, use both hydraulic and pneumatic operations to complete a task. For instance, a hydraulic system may be used to pick-up and transport a heavy object from one place to another; while during the same operation, a pneumatic system may be used to clamp the object in place during a specific machining operation. In addition, it is quite common for a smaller pneumatic system to control the valves of a larger hydraulic system.

One of the main differences between a hydraulic and pneumatic system is cost. Pneumatic systems are generally less costly since a common air source can be used to provide power for several systems. Also, pneumatic system components are less expensive to manufacture because they are not subject to the extreme pressures that are sometimes used in hydraulic systems. Finally, a pneumatic system uses atmospheric air to produce the required force, and, once this air has been used, it is vented back into the atmosphere. This characteristic negates the use of recirculating hydraulic fluids; thus, the possibility of abrasive contaminants building up in a system are almost nil.

## Pneumatic System Components

The simple pneumatic system, shown in Figure 2-33, is very similar to the basic hydraulic system just discussed. As we proceed with our discussion of the pneumatic system, you will see that many of the components have the same name and perform the same task as they did in the hydraulic system.

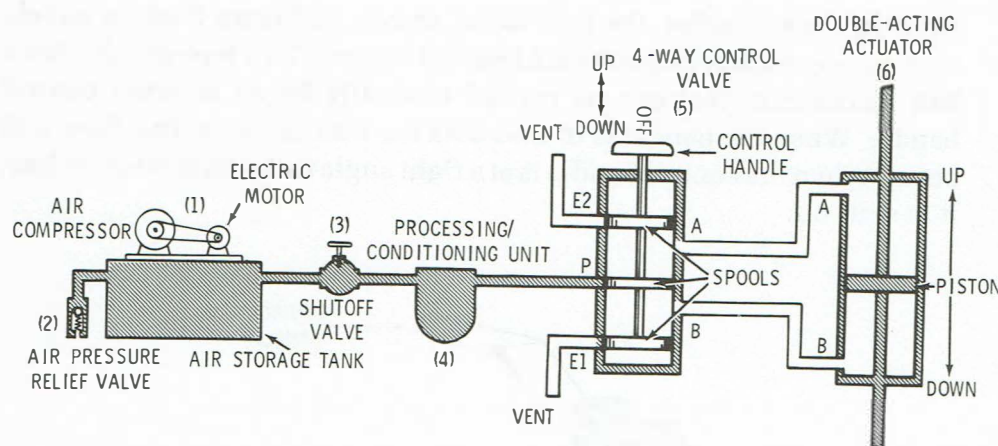


Figure 2-33  
Simple Pneumatic System

### AIR COMPRESSOR-MOTOR-AIR STORAGE TANK UNIT

The air compressor, motor, and air storage tank, Figure 2-33 (1), is usually a single unit with the compressor and the motor mounted on top of the air storage tank. The electric motor drives the air compressor, which in turn takes air from the atmosphere, compresses it, and routes it to the air storage tank. The storage tank acts as a reservoir for the compressed air, holding the air until it is needed by the system.

The air storage tank is equipped with a 1-way pressure relief valve (2) which prevents an excess pressure from building up in the storage tank. The pressure relief valve is set at a predetermined level; if the pressure inside the storage tank exceeds that level, the valve automatically opens, venting the excess pressure to the atmosphere.

A complete small unit can be mounted near the system it is serving, or, as is the case of large industrial applications, a much larger compressor and storage tank can provide power to several pneumatic systems. If several systems are being supplied from one source, pressure reducing valves would be required to supply each system with the desired pressure.

## SHUT-OFF VALVES

The shut-off valve, Figure 2-33 (3), is a simple 2-way directional control valve whose input and output connections are placed in series with the pneumatic transmission line. It is designed to provide full control; simply stated, it either permits air flow or shuts it off. The gate or globe valves discussed in the hydraulic section (Figure 2-24A and B), are two types of valves that are also used to provide flow control in pneumatic systems.

Another type of valve, the ball valve, shown in Figure 2-34, is widely used in pneumatic systems to achieve full control. This type of valve has a ball mechanism that can be rotated manually by an external control handle. When the handle is in line with the transmission line, flow will occur. When the control handle is at a right angle to the transmission line, it is shut off.

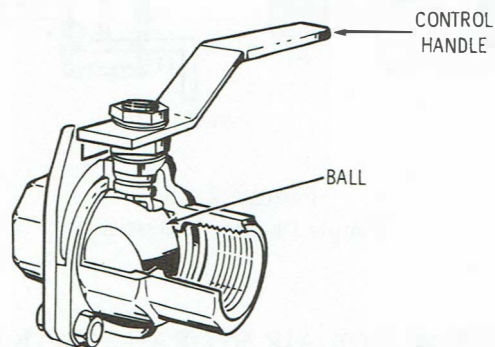


Figure 2-34  
Ball Type Two-Way Control Valve  
(shown in the off position)

A two-way control valve is simply an on-off control device; hence, it should not be used to regulate the amount of flow. When flow control is required, a valve similar to the one previously shown in Figure 2-27 should be used.



## PROCESSING AND CONDITIONING UNIT

The processing and conditioning unit, shown in Figure 2-33 (4), treats the compressed air before it is used by the system. This unit takes the compressed air from the storage tank and removes any dirt and moisture contained in the air. This is usually accomplished by an air filter with a condensation trap and drain. Once the compressed air has been cleaned and dried, it is treated with a fine mist of oil. The oil provides lubrication for all parts throughout the system. Many conditioning units also incorporate a pressure regulating valve in the unit to regulate air that is supplied to the rest of the system.

## CONTROL VALVES AND ACTUATORS

As you recall from our discussion of the hydraulic system, a control valve and an actuator work hand-in-hand to position a load. In the case of the simple pneumatic system shown in Figure 2-33, a 4-way control valve (5) is used to position a double-acting actuator (6).

In this illustration, when the control handle is placed in the down position, the valve spools also move down. This permits compressed air to flow from port P, through the control valve, out port A of the control valve, through the transmission line and into port A of the actuator. The compressed air forces the actuator piston down which in turn forces air out of the actuator through port B. Because of the position of the control valve spools, the air forced out of the actuator is allowed to flow into port B of the control valve, and out of port E1 to the atmosphere.

When the control handle is placed in the up position, the valve spools are also placed in the up position. This permits the compressed air to flow from port P, through the valve, out port B of the control valve, through the transmission line, and into port B of the actuator. This action forces the actuator piston to the up position. Again, because of the position of the valve spools, air is allowed to leave port A of the actuator, travel through the transmission line, pass through the control valve from ports A to E2, and vent into the atmosphere.

When the control handle is placed in the off position, as seen in the illustration, all control valve ports are blocked; thus, no work is accomplished by the actuator.

As in the hydraulic system, various types of control valves and actuators can be used, depending on the job to be performed.



## Programmed Review

39. Pneumatic systems are generally used to position relatively _____ loads. (heavy/light)
40. (light) A _____ is used to store compressed air in a pneumatic system.
41. (storage tank) A _____ valve prevents excess pressure from building up inside a storage tank.
42. (pressure relief) A ball valve is used to provide _____ control in a pneumatic system. (full/partial)
43. (full) Lubrication in a pneumatic system is accomplished by adding a fine mist of _____ to the compressed air.
44. (oil) In a pneumatic system, once the compressed air has been used it _____ normally returned to the storage tank. (is/is not)
(is not)

## EXPERIMENT

Perform Experiment 3. You will find this experiment in Unit 12. After you finish the experiment, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers, then select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct ones that appear after the examination.

1. Which component in the revolving-armature AC generator provides the stationary electro-magnetic field?
  - A. Rotor.
  - B. Field windings.
  - C. Slip rings.
  - D. Stator.
2. AC generators are rated at a specified frequency and power factor. The specified power factor is usually considered to be?
  - A. 80% lagging.
  - B. 80% leading.
  - C. 100% lagging.
  - D. 100% leading.
3. In the single-phase AC generator shown in Figure 2-5, which has four groups of series aiding stator windings and a four-pole rotor, the voltage induced in stator winding #3 would be:
  - A. In the opposite direction of the voltage induced in stator winding #1.
  - B. In the opposite direction of the voltage induced in stator winding #4.
  - C. In the same direction of the voltage induced in stator winding #2.
  - D. In the same direction of the voltage induced in stator winding #4.
4. In a wye-connected three-phase AC generator, each load is connected across:
  - A. Two phases in parallel.
  - B. One phase in parallel.
  - C. One phase in series.
  - D. Two phases in series.

5. In a three-phase, four-wire, wye-connected circuit, current flows in the neutral wire when:
  - A. Output voltage is increased.
  - B. Output voltage is decreased.
  - C. The loads are balanced.
  - D. The loads are unbalanced.
6. In a three-phase delta-connected AC generator line voltage is:
  - A. 1.73 times greater than single phase voltage.
  - B. Twice as great as single phase voltage.
  - C. Equal to single phase voltage.
  - D. One half the value of single phase voltage.
7. Which of the following type AC generator is capable of supplying three single-phase voltage outputs?
  - A. A wye-connected single-phase generator.
  - B. A delta-connected single-phase generator.
  - C. A wye-connected three-phase generator.
  - D. A delta-connected three-phase generator.
8. Which of the following is the most common method of increasing the output voltage of an AC generator?
  - A. Increasing the exciter output current.
  - B. Increasing the number of conductors in the stator.
  - C. Increasing the speed of rotation of the rotor.
  - D. Adding permanent magnets to the rotor core.
9. Which of the following AC generators is capable of supplying the largest output current?
  - A. Revolving-armature generator.
  - B. Delta-connected generator.
  - C. Wye-connected generator.
  - D. Revolving-stator generator.
10. One main advantage of a DC motor compared to an AC motor is the fact that:
  - A. DC motors are generally less expensive.
  - B. DC motors do not require brushes and commutators.
  - C. AC motors can not handle heavy loads.
  - D. DC motor speed is easier to control.

11. You can reverse direction of a three-phase induction motor by:
  - A. Removing one of the three input connections.
  - B. Shorting one of the three input connections to ground.
  - C. Interchanging any two input connections.
  - D. Three-phase induction motors cannot be reversed.
12. The synchronous speed at which an AC induction motor field rotates is:
  - A. Inversely proportional to the applied voltage frequency.
  - B. Inversely proportional to the number of stator coils.
  - C. Directly proportional to the applied voltage frequency.
  - D. Both B and C are correct.
13. Which of the following causes current to flow in the rotor of a three-phase induction motor?
  - A. The rotating stator field.
  - B. The DC exciter unit.
  - C. The torque of the rotor's load.
  - D. Permanent magnets mounted on the stator.
14. In a three-phase induction motor using a form-wound rotor, an external variable resistance is connected to the rotor to:
  - A. Produce a low starting torque.
  - B. Produce a high starting torque.
  - C. Maintain a constant motor running speed.
  - D. Provide the motor with the capability of reversing direction without reversing motor leads.
15. Which of the following is not a characteristic of an AC synchronous motor?
  - A. Rotating magnetic stator field.
  - B. Form-wound rotor.
  - C. Constant speed under all load conditions.
  - D. The synchronous motor starts as an induction motor.

16. In a split-phase induction motor, the starting winding and the main winding are:
- A. Connected in parallel across the supply line.
  - B. Displaced by 90 electrical degrees.
  - C. Physically contained in the motor's stator.
  - D. All of the above are correct.
17. In a single-phase AC capacitor motor, a capacitor is:
- A. Placed in parallel with the main winding.
  - B. Placed in series with the main winding.
  - C. Placed in series with the starting winding.
  - D. Placed in parallel with the starting winding.
18. Which of the following single-phase motors will provide the greatest amount of starting torque?
- A. Capacitor-start.
  - B. Split-phase.
  - C. Delta-connected stator.
  - D. Delta-connected rotor.
19. The baffle plate in a hydraulic reservoir is used to:
- A. Reduce turbulence inside the reservoir.
  - B. Get rid of trapped air in the system.
  - C. Act as a barricade to fluid contaminants.
  - D. All of the above are correct.
20. Which of the following type valves is considered to be a one-way valve?
- A. Gate valve.
  - B. Flapper valve.
  - C. Globe valve.
  - D. Spool valve.
21. Which type of valve is usually used to control a double-acting actuator?
- A. Pressure relief valve.
  - B. 3-way spool valve.
  - C. Flow control valve.
  - D. 4-way spool valve.



22. If a flow control valve is placed between the pump and the load it is referred as what type of operation?
- A. Run-away.
  - B. Meter-out.
  - C. Meter-in.
  - D. Free-regulation.
23. A double-acting cylinder with a piston rod of equal size on each side of the piston, is called?
- A. A nondifferential actuator.
  - B. An unbalanced cylinder.
  - C. A differential actuator.
  - D. A ram.
24. Which of the following is **not** a characteristic of a pneumatic system?
- A. It can position a heavier load than a hydraulic system.
  - B. It is less expensive than a hydraulic system.
  - C. Several systems can be run off a common air supply.
  - D. It is very reliable and accurate.
25. Which of the following is **not** a function of the air processing and conditioning unit?
- A. Removes dirt and moisture from the compressed air.
  - B. Provides lubrication for the pneumatic system.
  - C. Regulates the air to the rest of the system.
  - D. Stores the compressed air until it is required by the system.



## EXAMINATION ANSWERS

For your convenience, the page where the correct answer can be found is shown following the answer.

1. D — Stator. [ 2-8 ]
2. A — 80% lagging. [ 2-10 ]
3. B — In the opposite direction of the voltage induced in stator winding #4. [ 2-12 ]
4. D — Two phases in series. [ 2-15 ]
5. D — The loads are unbalanced. [ 2-15 ]
6. C — Equal to single phase voltage. [ 2-17 ]
7. C — A wye-connected three-phase generator. [ 2-18 ]
8. A — Increasing the exciter current. [ 2-19 ]
9. B — Delta-connected generator. [ 2-17 ]
10. D — DC motor speed is easier to control. [ 2-22 ]
11. C — Interchanging any two input connections. [ 2-25 ]
12. D — Both B and C are correct. [ 2-26 ]
13. A — Rotating stator field. [ 2-27 ]
14. B — Produce a high starting torque. [ 2-32, 2-33 ]
15. B — Form-wound rotor. [ 2-36 ]
16. D — All of the above. [ 2-38 ]
17. C — Placed in series with the starting winding. [ 2-39 ]
18. A — Capacitor-start. [ 2-40 ]

- 19. D — All of the above are correct. [ 2-45 ]
- 20. B — Flapper valve. [ 2-48 ]
- 21. D — Four-way spool valve. [ 2-53 ]
- 22. C — Meter-in. [ 2-54 ]
- 23. A — A nondifferential actuator. [ 2-59 ]
- 24. A — It can position a heavier load than a hydraulic system. [ 2-62 ]
- 25. D — Stores the compressed air until it is required by the system. [ 2-63, 2-64 ]

## *Unit 3*

# **DC POWER AND POSITIONING**



## CONTENTS

Introduction .....	3-3
Unit Objectives .....	3-4
Unit Activity Guide .....	3-5
Batteries .....	3-6
Nickel-Cadmium Batteries .....	3-10
Gelled-Electrolyte Batteries .....	3-31
DC Motors .....	3-52
DC Brushless Motors .....	3-61
Stepper Motors .....	3-69
Experiment .....	3-86
Unit Examination .....	3-87
Examination Answers .....	3-91

## INTRODUCTION

Many futuristic robotic applications will, no doubt, dictate that a robot have a greater degree of freedom of movement; thus, they will demand a power source that is portable, reliable, and renewable. In addition, future robots will require small, light-weight, easy-to-control, electric motors to move them about and provide a means of actuating their manipulator. This could very easily be accomplished by attaching an umbilical cord to the robot, but this method would place some restrictions on the robot's movements, thereby eliminating total freedom.

Because many robots of the future will be smaller than the industrial robots we previously discussed, domestic robots for instance, true freedom of movement is possible. This can be achieved through the use of rechargeable batteries and small, powerful, easily-controlled DC motors.

In this unit, we will discuss some of the batteries that could possibly be used to provide a reusable power source. Battery care and charging techniques will also be presented. We will then study how this power is used to operate and control the DC motors associated with mobile robots.

## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. State the differences between primary and secondary cells.
2. Explain the characteristics of a nickel-cadmium cell and how these characteristics would affect cell selection.
3. Define the various terms associated with nickel-cadmium cells.
4. Determine what charge rate to use when charging both nickel-cadmium and gelled-electrolyte batteries.
5. Determine what type of charging circuit is used with nickel-cadmium batteries.
6. Determine what type of charging circuit is used with gelled-electrolyte batteries.
7. State the advantages and disadvantages of series-wound, compound-wound, and shunt-wound DC motors.
8. Explain how motor load affects speed regulation.
9. Explain Hall-effect and how it is used to control a DC brushless motor.
10. Describe the operation of bipolar and unipolar permanent magnet stepping motors.
11. Describe the operation of a variable reluctance stepping motor.
12. Explain how bipolar and unipolar control circuits operate.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "Batteries."	_____
<input type="checkbox"/> Read "Nickel-Cadmium Batteries."	_____
<input type="checkbox"/> Answer "Programmed Review" Questions 1-12.	_____
<input type="checkbox"/> Read "Gelled-Electrolyte Batteries."	_____
<input type="checkbox"/> Answer "Programmed Review" Questions 13-23.	_____
<input type="checkbox"/> Read "DC Motors."	_____
<input type="checkbox"/> Answer "Programmed Review" Questions 24-31.	_____
<input type="checkbox"/> Read "DC Brushless Motors."	_____
<input type="checkbox"/> Answer "Programmed Review" Questions 32-37.	_____
<input type="checkbox"/> Read "Stepper Motors."	_____
<input type="checkbox"/> Answer "Programmed Review" Questions 38-48.	_____
<input type="checkbox"/> Perform Experiment 4.	_____
<input type="checkbox"/> Complete the "Unit Examination."	_____
<input type="checkbox"/> Check the "Examination Answers."	_____

## BATTERIES

Batteries are widely used as sources of direct-current electrical energy when other sources are not readily available or feasible. For example, the lead-acid battery in your automobile is the primary source of current for all electrical circuits. It can supply massive amounts of current to the engine starter motor while maintaining an adequate voltage for the ignition system, and it is rechargeable. However relatively inexpensive, lead-acid batteries do have their drawbacks; they are fairly heavy and are also prone to leakage or spillage of the electrolyte solution if they are not kept upright. The nickel-cadmium battery, on the other hand, can supply a large amount of electrical current, is fairly light-weight, can be used in virtually any position, and is also rechargeable; but, they are quite expensive. One happy medium between these two types of batteries is the gelled-electrolyte battery. But before we proceed to our discussion on nickel-cadmium and gelled-electrolyte batteries, we will present a refresher on the basic principles of batteries.

A battery consists of a number of cells assembled in a common container and connected together to function as a source of electrical power.



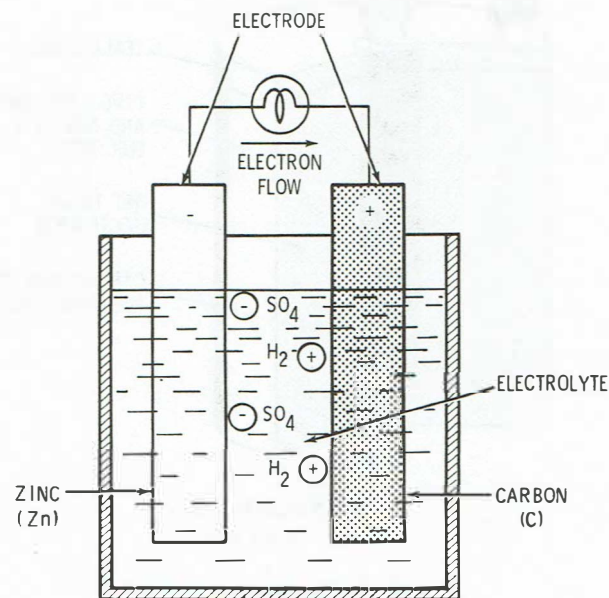


Figure 3-1  
Simple Voltaic Cell.

## Cell

A cell is a device that transforms chemical energy into electrical energy. The simplest cell, the voltaic cell, shown in Figure 3-1, consists of a piece of carbon (C) and a piece of zinc (Zn) suspended in a jar that contains a solution of water ( $\text{H}_2\text{O}$ ) and sulfuric acid ( $\text{H}_2\text{SO}_4$ ).

The cell is the fundamental unit of the battery. A simple cell consists of two strips, or electrodes, placed in a container that holds the electrolyte.

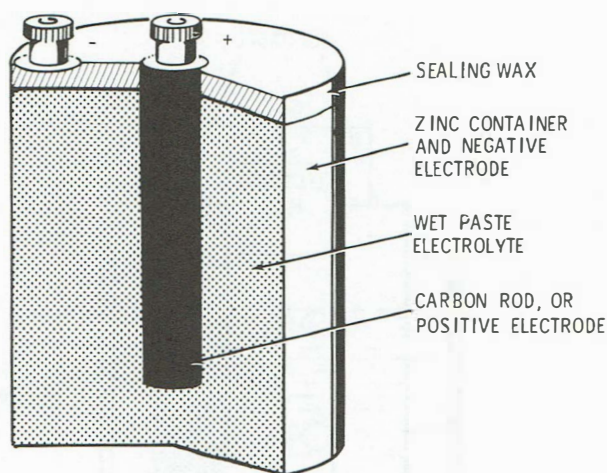


Figure 3-2  
Dry Cell.

## Electrodes

The electrodes are the conductors by which the current leaves or returns to the electrolyte. In the voltaic cell, they are carbon and zinc strips that are placed in the electrolyte; while in the dry cell, shown in Figure 3-2, they are the carbon rod in the center and the zinc container in which the cell is assembled.

## Electrolyte

The electrolyte is the solution that acts upon the electrodes which are placed in it. The electrolyte may be a salt, an acid, or an alkaline solution. In the simple voltaic cell, in Figure 3-1, and in the automobile storage battery, the electrolyte is a liquid; while in the dry cell, shown in Figure 3-2, and the gelled-cell, the electrolyte is a paste.

## Primary Cell

A primary cell is one in which the chemical action erodes one of the electrodes, usually the negative electrode. When this happens, the electrode must be replaced or the cell must be discarded. In the galvanic cell, the zinc electrode and the liquid electrolyte solution are usually replaced when this happens, but in the case of the dry cell, it is usually cheaper to buy a new cell. Most primary cells are not intended for recharging.

## Secondary Cell

A secondary cell is one in which the electrodes and the electrolyte are altered by the chemical action that takes place when the cell delivers current. These cells may be restored or recharged to their original condition by forcing an electric current through them in the **opposite** direction to that of current delivery or discharge. The automobile storage battery is a common example of a secondary cell.

## Battery

As was previously mentioned, a battery consists of two or more cells placed in a common container. The cells may be connected in series, in parallel, or in some combination of series and parallel, depending upon the amount of voltage and current required of the battery.

Since most of us are already familiar with the lead-acid batteries used in automobiles, we will direct our studies to the nickel-cadmium and gelled-electrolyte batteries. There are several other types of rechargeable batteries on the market today, but nickel-cadmium and gelled-electrolyte batteries are the most common.

## NICKEL-CADMIUM BATTERIES

Nickel-cadmium batteries are far superior to lead-acid batteries and are rapidly replacing them in many situations requiring renewable portable power. Some of the more prominent features of nickel-cadmium batteries are:

- The sealed types are virtually maintenance free; they contain no free electrolyte, and they can be operated and charged in any position, without damage to the battery or equipment.
- Nickel-cadmium batteries maintain an almost constant voltage throughout most of their discharge period. In addition, the voltage level varies only slightly with differing discharge rates. The nominal discharge voltage (voltage output of a cell or battery under load during discharge) per cell is 1.25 volts at room temperature, 68°F (20°C). Due to their low internal resistance and ability to maintain a constant discharge voltage, nickel-cadmium batteries are especially suited to high discharge or pulse current applications.
- Nickel-cadmium batteries can be recharged at high rates under controlled conditions. Many batteries can be charged in 3 to 5 hours without special controls or precautions.
- Nickel-cadmium batteries have the ability to withstand continuous overcharging at recommended rates and temperatures with no noticeable affect on battery life unless the charge rate exceeds the design limitations of the cell.

- Nickel-cadmium batteries are designed to operate over a wide temperature range with no appreciable effect on their output. This temperature range can vary from  $-40^{\circ}\text{F}$  to  $140^{\circ}\text{F}$  ( $-40^{\circ}\text{C}$  to  $60^{\circ}\text{C}$ ) for normal cells, with some specially constructed batteries capable of operating in extreme ambient conditions.
- Most nickel-cadmium batteries have a useful life of 300 to 1000 cycles of discharge (the number of times a battery can be discharged and recharged). By not completely discharging the battery and by reducing the amount of overcharge and heat presented to it, you will greatly extend its life.
- At room temperature, nickel-cadmium batteries will retain approximately 50% of their charge for 3 months. This self-discharge rate will increase at higher temperatures.
- Nickel-cadmium batteries can be stored for prolonged periods of time in either a charged or discharged condition without a significant, irreversible decline in their performance. After long periods of storage, they may require a few charge/discharge cycles to restore full capacity.



## Construction

The nickel-cadmium cell plates are constructed of nickel powder “sintered” (heat bonded) to a nickel wire screen. The active materials, nickel-hydroxide on the positive plate and cadmium-hydroxide on the negative plate, are electrically bonded to the basic plate structure. The separators are constructed of plastic, nylon cloth, or a special type of cellophane, and assembled as a cell core with plates. See Figure 3-3.

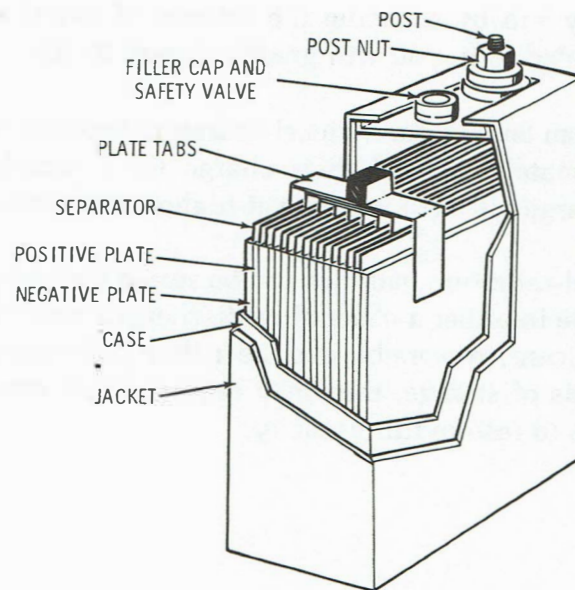


Figure 3-3  
Large Nickel-Cadmium Cell Using Liquid Electrolyte.

The construction of the sintered-plate cell is accomplished by a powder metallurgy process. Carbonyl nickel powder is lightly compressed in a mold and then is subjected either to a temperature of about 1600°F in a sintering furnace or to a sudden large electrical current. Either process causes the individual grains of nickel to weld at their points of contact, producing a porous film which is approximately 80% open holes and 20% solid nickel. These films are then impregnated with active materials. This impregnation of active materials is accomplished by soaking the films in a solution of nickel salts to make the positive plates, and a solution of cadmium salts to make the negative plates. The bath is completed when the films contain the amount of active material necessary to give them the capacity desired. Once the films have been impregnated, they are classified as plates.

The electrolyte used in a large nickel-cadmium cell is a 30-percent-by-weight solution of potassium hydroxide in distilled water. Chemically speaking, this is just about the exact opposite of the diluted sulfuric acid used in the lead-acid cell. As with lead-acid cells, there are limitations on the concentration of electrolyte solution that can be used in nickel-cadmium cells. That is, the specific gravity of the solution should not be outside the range of 1.240 to 1.300 at 70°F (21°C). The electrolyte in the nickel-cadmium cell does not chemically react with the plates as does the electrolyte in the lead-acid cell. It acts only as a conductor of current between the plates; therefore, no flaking or erosion of the active material takes place. Consequently the plates do not deteriorate, nor does the specific gravity of the electrolyte change appreciably. For this reason, it is not possible to determine the charge state of a nickel-cadmium cell by checking the electrolyte with a hydrometer; neither can the charge be determined by a voltage test because of the inherent characteristic that the voltage remains constant during 90 to 95 percent of the discharge cycle.

No external vent is required, since gassing of this type of cell is practically negligible. As a safety precaution, however, a safety valve is installed in the fill hole cap of each cell, as shown in Figure 3-3. This safety valve is designed to release any excess gas that is formed when the battery is improperly charged.

Unlike the previously discussed large nickel-cadmium cell, the cylindrical cell, shown in Figure 3-4, is a completely sealed unit. The cylindrical cell is constructed from six major components: a positive and negative electrode, a porous separator, electrolyte, a steel jacket, and a seal with a built-in safety vent.

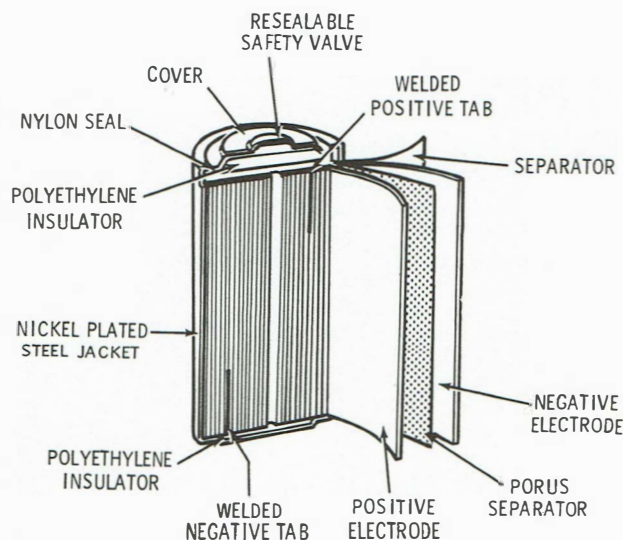


Figure 3-4  
Cylindrical Nickel-Cadmium Cell.

The positive and negative electrodes are manufactured in much the same manner as the plates of the larger nickel-cadmium cell. One major difference is that the electrodes are further processed, cut to size, and, with a separator between them, are coiled and assembled into cells.

The separator is usually an unwoven nylon material, highly absorbent to the alkaline electrolyte solution, and permeable to oxygen. The cell case is a nickel-plated steel jacket, and all internal connections are usually welded.

The cell is completed by sealing it with a positive seal, which incorporates a safety vent to prevent cell rupture in case of excessive gas pressure within the cell. Excessive internal pressure build-up can be caused by extreme charge or discharge rates.

The safety vent may be of either the hermetic or resealable type. Both are designed to release excessive internal pressure well below the point where the cell might normally rupture. Regardless of which type of vent is used, repeated venting of the cell will release gas; thus, lowering the electrolyte level within the cell. This, in turn, will cause the cell to dry out, decrease its performance, and ultimately fail to deliver power at the proper level. Whether it is hermetic or resealable, the vent's primary function is to provide a safety valve; this prevents accidental cell rupture caused by abuse or misuse, which in turn, could damage expensive electronic equipment.

## Operation of Nickel-Cadmium Cells

The electro-chemical reaction of the nickel-cadmium sealed cell differs greatly from the vented secondary cell.

After an open or vented type of cell is completely charged, both oxygen and hydrogen gases, as well as electrolyte fumes, will be present. These gases are vented through a valve or filler cap in the top of the cell, and are extremely explosive.

In the nickel-cadmium sealed cell, such gases caused by overcharging are prevented within the cell. This is due to the state of charge of the negative electrode at the time the cell is sealed. The negative electrode never becomes fully charged; thus, the emission of hydrogen gas is suppressed.

When the positive electrode reaches full capacity during charging, oxygen will be produced. The oxygen is channeled through the porous separator to the negative electrode, and oxidizes the metallic cadmium, causing cadmium-hydroxide to be produced. At the same time, the cadmium-hydroxide formed in this manner is continuously reduced by electro-chemical action back to metallic cadmium.

A balanced oxygen pressure is set up inside the cell, with the rate of evolution of oxygen gas equal to the rate of recombination with the metallic cadmium. The level of pressure within the cell is determined by the charge rate and is normally 7-15 PSI.

## REVERSAL PROTECTION

When three or more cells are connected in series to obtain a higher voltage, the possibility of a condition known as “cell reversal” exists. Cell reversal can take place during discharge when one of the cells, which may be slightly lower in capacity than the others, is driven to zero potential and then into reverse. During a reversal, hydrogen gas, normally evolved by the negative electrode, may be evolved by the positive electrode. In addition, oxygen gas, normally produced by the positive electrode, may be produced by the negative electrode. This phenomenon can damage the cell; therefore, most nickel-cadmium cells are constructed with reversal protection.

Reversal protection causes the evolution of hydrogen gas from the positive electrode to be suppressed, and the oxygen gas produced by the negative electrode during reversal to be recombined with the positive electrode. At moderate discharge rates, nickel-cadmium cells can be driven into reverse without permanently damaging the cell. However, you should avoid prolonged, deep, or frequent reversals, as they tend to shorten the life of the cell.

If you anticipate frequent, high-rate discharges, you should consider a warning or cut-off circuit to prevent a possible low cell from going below  $-0.2$  volts.



## General Characteristics

Figure 3-5, shows the three types of standard nickel-cadmium cells; rectangular, button and cylindrical, with the cylindrical being the most common and widely used cell. These cells have some common characteristics, or parameters if you prefer, which are noted below:

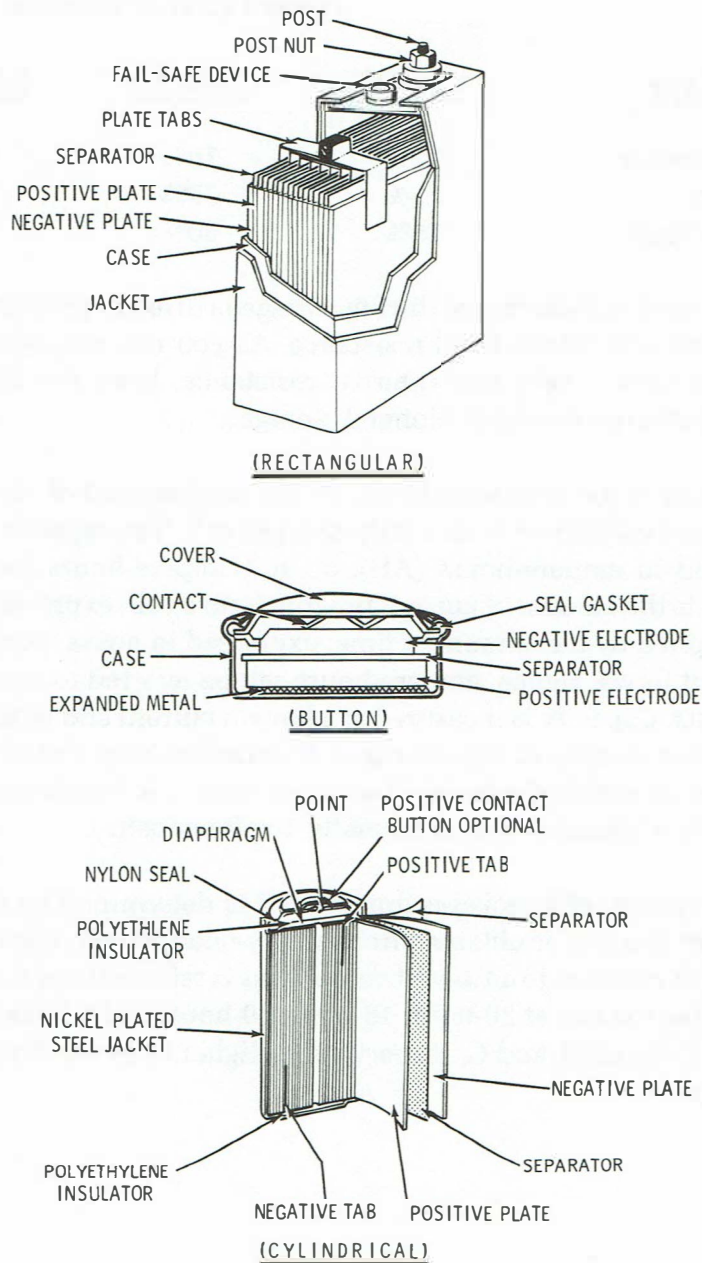


Figure 3-5  
Standard Types of Nickel-Cadmium Cells.

**Charge Retention** — is the ability of a cell to retain a charge while in storage and is based on a storage temperature of 68°F (20°C). Cells stored at a higher temperature will have a decrease in charge retention; conversely, cells stored at a lower temperature will have an increase in charge retention. All cells that have been stored should be charged prior to use to restore full capacity.

**Percent Charge Retained**

<u>Cell Type</u>	<u>1 Month</u>	<u>3 Months</u>	<u>5 Months</u>
Rectangular	75%	70%	63%
Button	75%	70%	63%
Cylindrical	75%	50%	15%

The rate of self-discharge during storage is directly proportional to temperature and internal cell resistance. As you can see, cylindrical cells, which have a very low internal resistance, have the highest rate of self-discharge during prolonged storage.

**Capacity** is the term used to define the total amount of electrical energy that can be obtained from a fully-charged cell. The capacity of a cell is expressed in ampere-hours (AH), or milliampere-hours (mah). Ampere-hours is the amount of current flowing from a cell, expressed in amperes, multiplied by the amount of time, expressed in hours, during which the current flows. Hence, ampere-hours can be referred to as a current-time product. Capacity is measured at a known current and temperature, for a specified amount of time to a specific cutoff voltage. Cutoff voltage is the voltage at which discharge of a cell or battery is terminated. For nickel-cadmium batteries, this is normally 1 volt per cell.

The capacity of a nickel-cadmium cell is determined by the amount of current that can be obtained from a fully-charged cell, discharged at 68°F (20°C) for 5 hours to a 1.0-volt cutoff. This is referred to as the C/5 rate. Discharging the cell at 20-hour, 15-hour, 10-hour, and 1-hour rates is called C/20, C/15, C/10, and C, respectively. Higher rates are designated as 2C, 3C, etc.

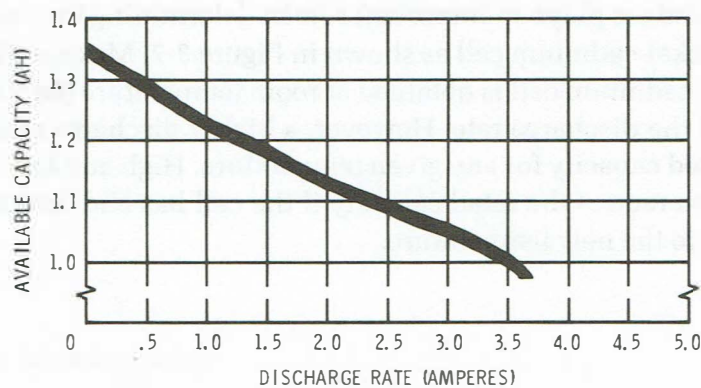


Figure 3-6  
Capacity vs Discharge Rate.

The capacity of a nickel-cadmium cell, to a specified cutoff voltage, and at a specific temperature, decreases as the discharge rate increases. This is illustrated in Figure 3-6.

**DISCHARGE RATE** — The current at which a cell or battery is discharged. Frequently expressed as a function of its rated capacity. Example: Discharge of a 6.0AH cell over a 5-hour period of time would equal a discharge rate of 1.2 amperes, or 1200 milliamperes.

**Discharge Rate = Ampere Hours/Time, or 6.0AH/5-Hours = 1.2 amperes or 1200 milliamperes.**

“Rated Capacity” is another term you will encounter when selecting or using batteries. Be sure you do not confuse the terms “rated capacity” and “capacity” when dealing with batteries. To refresh your memory, **capacity** is the total electrical energy available from a fully-charged cell or battery. Whereas, **rated capacity** is defined as the conservative estimate of the amount of capacity that can be drawn from a fully-charged cell or battery when discharged at a specific rate, at a known temperature to a specific cutoff voltage.

Temperature plays an important role in determining the overall capacity of a nickel-cadmium cell as shown in Figure 3-7. Maximum capacity of a nickel-cadmium cell is obtained at room temperature (68°F/20°C) regardless of the discharge rate. However, a higher discharge rate does reduce the rated capacity for any given temperature. High and low temperatures can also reduce the rated capacity if the cell has had time to absorb and adjust to the new temperature.

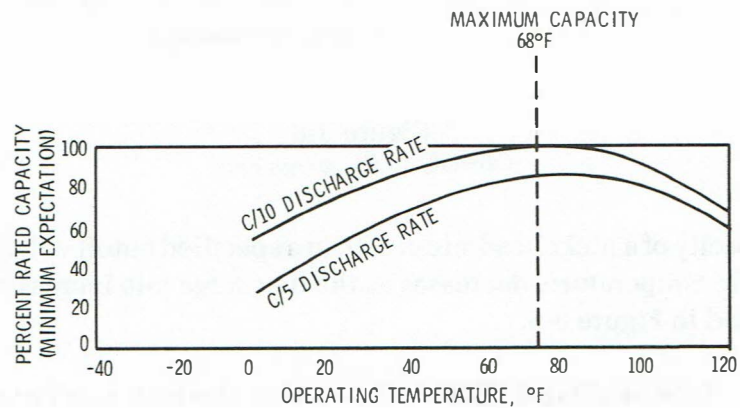


Figure 3-7  
Capacity vs Temperature.

## Charging Characteristics And Charging Techniques

One term you must become familiar with in respect to battery charging is "charge rate." Charge rate is the current at which a cell or battery is charged, expressed as a function of rated capacity. For example, the 3-hour rate for a 1.5 AH cell equals  $1.5/3 = 0.5$  amps or 500 milliamps. This is also called the C/3 rate.

Nickel-cadmium cells have the capability of withstanding continuous overcharging at a constant current throughout the range of C/20 to C/3 for cylindrical cells, and C/50 to C/10 for the smaller button cells. However, while button cells can accept an overcharge, they should not be overcharged for extended periods of time.

For cylindrical nickel-cadmium cells, chargers for the range of C/20 to C/3 are relatively simple. Automatic termination or reduction of the charge is not required; this is due to the ability of the cell to accept a continuous overcharge without causing damage to the cell. When charge rates greater than C/3, commonly referred to as rapid or fast-rate charging, are used, the charge rate must be monitored and terminated when the cell has reached its capacity. You can monitor the charge rate by observing one or more of the following cell parameters: temperature, voltage, or pressure.

### TRICKLE CHARGE

A cell or battery is trickle charged so that it will retain its full charge. A trickle charge is generally applied to a cell or battery whose primary function is to provide standby or emergency power to electronic equipment. Many computer systems use nickel-cadmium batteries that have a trickle charge constantly applied to them to retain their memory in case of power failure. The trickle charge rate is not sufficient to efficiently charge the cell or battery, and it is used only after the cell or battery has been fully charged. For standby operations, a minimum charge is preferred.



### MINIMUM CHARGE

A minimum charge for nickel-cadmium cells is normally between C/20 and C/10. This charge rate is sufficient to bring a cell up to a fully charged condition and to maintain it in this state. A charge rate of C/15 is considered ideal, as it minimizes the heating effect during overcharge; thus prolonging cell life. A minimum charge should be kept on a mobile robot when it is not in use; full power would then be available when the robot is called upon to perform a task.

### STANDARD CHARGE

The standard or normal charge for a nickel-cadmium battery is a 14-hour or "overnight" charge, using constant current, at the C/10 rate. This will bring a battery to the fully-charged condition. The need to charge a fully-discharged battery for more than 10 hours is due to normal charging inefficiencies inherent to secondary cells. A cell can be overcharged at the C/10 rate for long periods of time at room temperature without causing damage to the cell. Again, button cells are not designed for prolonged overcharge at these rates. The C/10 rate is normally used when cyclic operation is the norm. That is, when charge and discharge cycles take place at regular intervals. That would be the case of a mobile robot that was used during the day and put on a standard charge at night.

### QUICK CHARGE

For operations requiring 2 or 3 full cycles in a 24-hour period, quick-charge nickel-cadmium cells are available. These cells are designed to accept an overcharge current up to the C/3 rate, and will recharge in 3 to 5 hours, depending on the charger and the cell. Cells of this type are ideal for equipment that has to be used several times during a 24-hour period.

### RAPID CHARGE

Rapid or fast charging is for those applications that require recharging in less than 3 hours (greater than the C/3 rate). This method of charging requires a controlled charge circuit. The controlled charge circuit is necessary because of the high charge rates, and the heat and gas generated during overcharge. As previously stated, some type of temperature, voltage, or pressure monitoring is required when using the rapid charging method.

## CHARGING PARAMETERS

Figure 3-8, shows a typical constant-current battery charger using half-wave rectification. As a guide for constructing this basic charging circuit, components should be selected as follows:

1. Transformer T1 secondary voltage ( $E_s$ ) should be 2 to 2.5 times the nominal voltage of the battery or cell being charged.
2. Rectifier D1 should have a peak inverse voltage (PIV) of at least twice the voltage of the transformer secondary winding.
3. Resistor R1 should have a rated wattage that is at least equal to the product of the charging current and the secondary voltage of transformer T1.  
(resistor wattage =  $E_s \times I_{chg}$ )
4. R1 ohmic value is computed to reduce the charging current to the specified value for the battery.

For example, to charge a 6-volt, 2.4AH battery at the C/3 rate of 800 ma:

Nominal Voltage = 6 volts.

Transformer Secondary:  $2.5 \times 6 \text{ V} = 15 \text{ volts, minimum.}$

Rectifier PIV =  $2 \times 15 \text{ V} = 30 \text{ volts, minimum.}$

Resistor Wattage =  $15 \text{ V} \times 800 \text{ ma} = 12 \text{ watts minimum.}$

Resistor Value =  $15 \text{ V} - 6 \text{ V} / 800 \text{ ma} = 11.25 \text{ ohms, minimum.}$

Charge Time — 3.5 to 4 hours minimum, at room temperature.

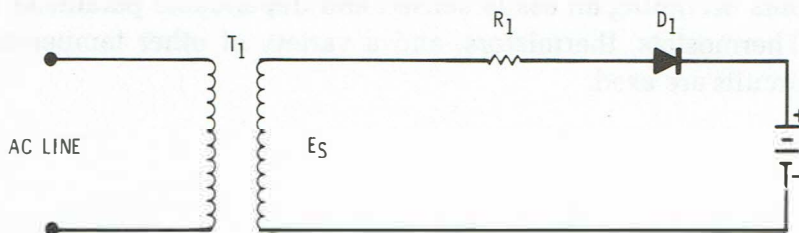


Figure 3-8  
Constant Current Half-Wave Rectification Battery Charger.

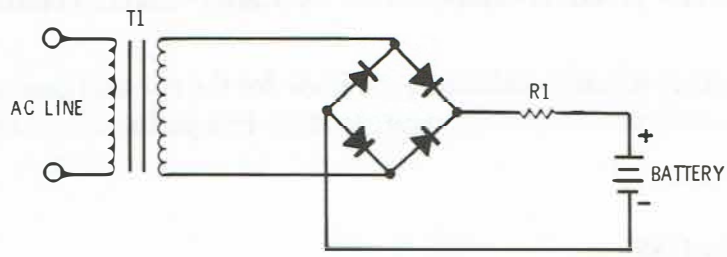
### ALTERNATE CHARGING CIRCUITS

Full-wave rectification is generally used to charge higher capacity cells, or when higher charge rates are desired. The half-wave rectifier charger is very popular, but it is not efficient for these purposes.

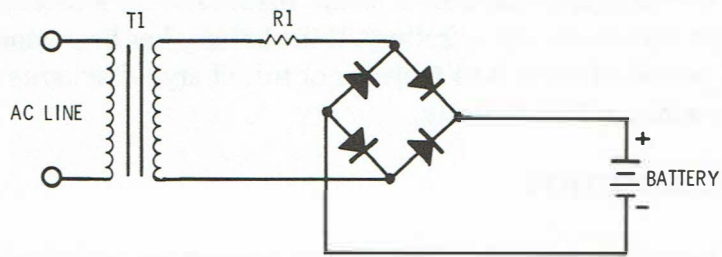
Figure 3-9A, shows a full-wave rectifier with the resistor between the battery and the rectifier. This configuration is known as a constant current charger when the resistor is placed between the transformer and the rectifier. Figure 3-9B is referred to as a variable current, modified constant voltage charger. The charging resistor limits the value of current that will flow. With low resistance the charging current will be high, and with high resistance the charging current will be low. Since the difference between battery voltage between full charge and discharge is relatively small in comparison with the charging voltage, the charging current will decrease very little as the battery nears full charge.

When rapid charging of a nickel-cadmium battery is required, cell voltage, temperature, or pressure must be monitored. As you recall, this monitoring is necessary to reduce or terminate the charge rate once the battery reaches full charge.

Cell voltage is very sensitive to charge rate as well as ambient conditions, and becomes a difficult variable to monitor unless the ambient is known. Cell pressure is also difficult to monitor, which leaves us with temperature monitoring, the most common method of control. The temperature rise of the cell in overcharge is practically independent of ambient temperature, thus becoming an easily sensed and dependable parameter for control. Thermostats, thermistors, and a variety of other temperature sensing circuits are used.



(A)  
CONSTANT CURRENT CHARGER



(B)  
VARIABLE CURRENT - VOLTAGE CHARGER

**Figure 3-9**  
Chargers For Charging Higher Capacity Cells:  
(A) Constant Current Charger,  
(B) Variable Current-Modified Voltage Charger.

## Care And Maintenance of Nickel-Cadmium Cells

The following recommendations are made for the use and care of nickel-cadmium cells or batteries, to insure trouble-free performance and longer life.

### PRIOR TO USE

Nickel-cadmium batteries, like most other electronic components, are usually supplied with manufacturers specification sheets. Study these data sheets to make sure the battery you have chosen will meet your requirements. Before you first use the battery, you should check it for capacity by discharging the battery at the C/10 rate to 1.0 volt per cell, and then fully recharging it according to the manufacturer's recommendations for that particular cell or battery. If the battery has been stored for a prolonged period of time, 3 to 5 cycles of full charge-discharge may be required to achieve full capacity.

### CIRCUIT CONNECTION

If you use the battery in applications requiring medium or high discharge rates, you should use welded, crimped, or soldered connections. For pressure connections, make sure they have a firm contact on clean surfaces. Do not solder directly to the cell. Use cells with solder lugs to avoid damage.



## NICKEL-CADMIUM USE

Nickel-cadmium cells or batteries should be used in series; connecting them in parallel is not recommended. Also, avoid discharges into deep reversal, as previously discussed. Cut-off at 1.0 volt per cell is recommended.

## CHARGING

The charging characteristics and charging methods previously discussed cover the general scope for charging nickel-cadmium batteries. Note that constant current, or modified constant current charging is the preferred method. Avoid constant voltage charging, unless close regulation and overcurrent protection can be provided.

## GENERAL

Be careful when you handle nickel-cadmium batteries to prevent accidental short circuiting. Because of their extremely low internal resistance, these batteries will discharge at extremely high current rates and produce very high temperatures when shorted, which can damage the cell, and could cause personal injury.

Nickel-cadmium batteries should be kept cool, as this will help improve battery life and performance. They should not be placed near other heat producing devices and, if possible, should be ventilated when they are used in high rate or high temperature applications.

After a battery has completed many charge and discharge cycles, a white powdery deposit may appear in the seal area. This is carbonate deposit, and is considered harmless to the battery and other components.

If cell replacement becomes necessary in a nickel-cadmium battery, the battery should be charged and cycled in the same manner as if it had been stored for a long period of time. This assures that the battery will be charged to its full capacity prior to use.

Before ending our discussion of nickel-cadmium cells, we will briefly cover cell failures. There are two types of cell failures — reversible and permanent.

### **REVERSIBLE FAILURES**

A reversible failure is one where the condition causing the failure can be eliminated and the cell can be restored to active use. Failures of this nature are usually due to shallow charge/discharge cycles, which causes the cell to appear as if it has lost capacity. This condition is sometimes referred to as the “memory” effect. It seldom happens, but if it does, you can remove the memory effect with a deep discharge, then a full recharge, of the cell.

A similar loss of capacity can result from extended overcharging of a cell or battery. Again, should this occur, you can restore full capacity by a full discharge/recharge cycle.

### **PERMANENT FAILURES**

A permanent failure is one where the cell is no longer usable and has to be replaced. Failures of this type are generally caused by time, temperature, rate and depth of discharge, and the circuit application of the cell or battery. The failure is usually the result of an internal short or an open circuit within the cell.

A cell or battery is considered to have permanent failure if it no longer operates the device for which it was designed, due to a reduction in capacity. The battery, however, may be usable in other applications requiring less battery capacity.

## Programmed Review

1.	A cell is a device that transforms _____ energy into electrical energy.
2.	(chemical) When recharging a secondary cell, current is forced through the cell in the _____ direction to that of current discharge. (same/opposite)
3.	(opposite) The nickel-cadmium cell _____ have the (does/does not) ability to withstand continuous overcharging at recommended rates and temperatures.
4.	(does) The state of charge of a large, liquid electrolyte, nickel-cadmium cell _____ be determined with a hydrometer. (can/cannot)
5.	(cannot) A phenomenon known as _____ takes place when one cell of a nickel-cadmium battery is driven to zero potential and then into reverse.
6.	(cell reversal) Internal cell _____ greatly effects the rate of self-discharge of a nickel-cadmium cell, during storage.

7. (resistance) The \_\_\_\_\_ of a cell is determined by the total amount of electrical energy that can be obtained from a fully charged cell.

8. (capacity) Maximum capacity of a nickel-cadmium cell is obtained at \_\_\_\_\_ temperature regardless of the discharge rate.

9. (room or 68 degrees) A trickle charge is generally applied to a nickel-cadmium battery whose primary function is to provide \_\_\_\_\_ power to electronic equipment.

10. (standby or emergency) A constant \_\_\_\_\_ is used to charge a nickel-cadmium battery overnight.

11. (current) The \_\_\_\_\_ method of recharging a nickel-cadmium cell requires a controlled charging circuit.

12. (rapid or fast) Prior to first using a nickel-cadmium battery, it is recommended that you check the \_\_\_\_\_ of the battery.

(capacity)

## GELLED-ELECTROLYTE BATTERIES

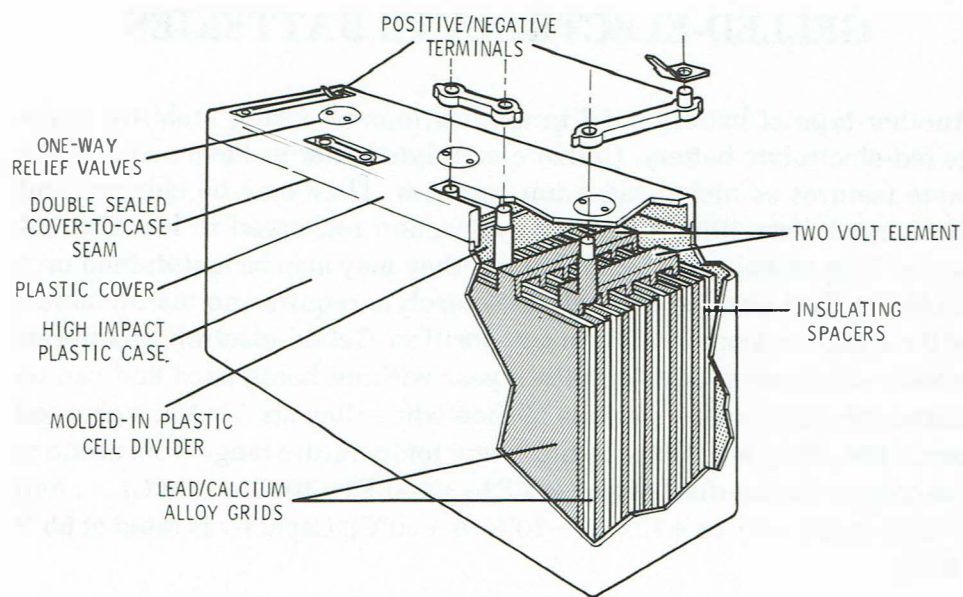
Another type of battery used in applications requiring mobility is the gelled-electrolyte battery. Gelled-electrolyte batteries have many of the same features as nickel-cadmium batteries. They may be charged and discharged from 100 up to 1000 cycles, and recharged in less than 14 hours. Like nickel-cadmium batteries, they may also be maintained on a trickle or float charge. The gelled-electrolyte requires no maintenance, will not leak and can be used in any position. Gelled-electrolyte batteries remain chargeable for more than a year without being used and can be stored for much longer periods of time when they are kept in a charged condition. They will also operate in any temperature range from arctic to sub-tropic; during discharge:  $-40^{\circ}\text{F}$  to  $+140^{\circ}\text{F}$  ( $-40^{\circ}\text{C}$  to  $+60^{\circ}\text{C}$ ) . . . and charge: from  $-4^{\circ}\text{F}$  to  $+122^{\circ}\text{F}$  ( $-20^{\circ}\text{C}$  to  $+50^{\circ}\text{C}$ ). Capacity is rated at  $68^{\circ}\text{F}$  ( $20^{\circ}\text{C}$ ).

Due to their design, gelled-electrolyte batteries have the ability to recover from deep discharges if they become totally discharged because of a power failure or by a switch accidentally being left on.

The gelled-electrolyte battery's 2.10 to 2.20-volts-per-cell open-circuit voltage is very high compared to the nickel-cadmium's 1.2 volts. In addition, gelled-electrolyte batteries have no memory problem; thus, they can deliver their rated capacity no matter what the previous usage history.

Finally, gelled-electrolyte batteries, unlike silver, nickel, or cadmium batteries, use lead dioxide, which is less expensive to produce and far more readily available.





**Figure 3-10**  
Gelled-Electrolyte Battery.

## Construction

Figure 3-10, shows a cut-away view of a gelled-electrolyte battery. You will notice that the construction is much the same as the standard lead-acid battery used in automobiles. Cell elements are constructed with pasted plates. Moderately thick lead/calcium grids are used to ensure optimum life, and the electrode surface area is as large as possible.

The gelled-electrolyte produces a network of porous paths between the positive and negative plates. The oxygen being evolved from the positive plates does not escape the sealed container but uses the porous paths to travel to the negative plates, and combines with hydrogen to produce water. All elements are sealed within an airtight, high-impact polystyrene case that is provided with a safety relief valve.

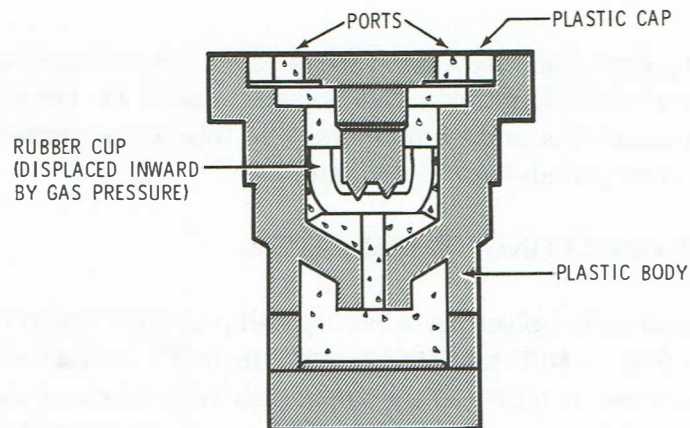


Figure 3-11  
Safety Relief Valve.

The safety relief valve, shown in Figure 3-11, provides safe operation in the event of gas build-up within the cell due to extreme overcharge or temperature variations. The relief valve will expell the gases when internal pressure reaches a preset value.

Components are plastic and rubber. The rubber cup in the center is held in position by a cap containing ports to the atmosphere. The outside atmosphere cannot enter the valve. An internal gas pressure forces the walls of the cup inward, allowing the gas to escape through the ports; then the cup automatically reseals.

## Gelled-Electrolyte Parameters

Each type of gelled-electrolyte battery has its own electrical characteristics; therefore, manufacturer's data sheets should be consulted for specific characteristics and parameters. The following parameters are common to most gelled-electrolyte batteries.

### DISCHARGE OPERATING TEMPERATURE

A fully-charged gelled-electrolyte battery will provide a useful output in any climate from  $-40^{\circ}\text{F}$  to  $140^{\circ}\text{F}$  ( $-40^{\circ}\text{C}$  to  $60^{\circ}\text{C}$ ). Gelled-electrolyte batteries are rated at  $68^{\circ}\text{F}$  ( $20^{\circ}\text{C}$ ). Capacities will increase above this temperature and decrease below this temperature. This can be seen in Figure 3-12.

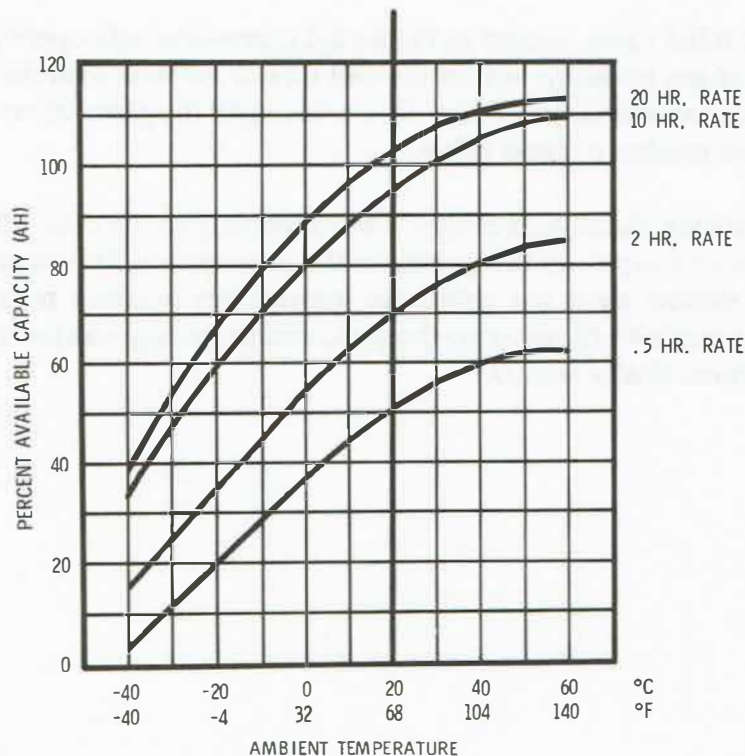


Figure 3-12  
Capacity vs Temperature.

## STORAGE

Both service life and storage, or shelf, life are greatly affected by temperature. The general statement of a 50% reduction in life for every increase in temperature of 18°F (10°C) above room temperature can be used as a guide line. Storage, or shelf, life can be increased by storing the batteries at lower temperatures. Batteries must be stored at temperatures above -40°F (-40°C) to prevent freezing of the electrolyte solution. For longest life, batteries should be fully charged before they are stored.

In addition, batteries stored at 50°F (10°C) or lower should be recharged at least once a year. Batteries stored at temperatures from 50°F to 80°F (10°C to 27°C) should be charged every six months, or every three months in temperatures from 80°F to 100°F (27°C to 38°C).

## CELL REVERSAL

Unlike the nickel-cadmium batteries, gelled-electrolyte batteries can be totally discharged and no permanent cell reversal will occur. However, to obtain maximum battery life, a cutoff voltage of 1.75 volts per cell is recommended.

## BATTERY LIFE

Most gelled-electrolytic batteries have a design life of four years when they are connected to a float charger and are used under normal conditions. The life of a battery will vary according to the depth of discharge and the temperature conditions it encounters. Charge-discharge life for these batteries is normally between 100 and 1000 charge-discharge cycles. Charge time is normally less than 14 hours.

Gelled-electrolyte batteries will have a longer life when discharged at rates of C/5 or less. That is, a battery discharged at a C/5 rate will normally have longer life than a battery discharged at a C/1 rate. When discharged at faster rates, which in turn creates more internal heat, more of the active material is consumed and the grids eventually become inactive and brittle; thus, shortening battery life.

## Capacity

The term “capacity”, as applied to gelled-electrolyte batteries, is the same as for nickel-cadmium batteries. To refresh your memory, **capacity** is the total amount of electrical energy that can be obtained from a fully-charged cell. The capacity of a cell is expressed in ampere-hours (AH) or milliampere-hours (mah), which is a current-time product. Again, the capacity value of a cell is dependent upon the discharge current, temperature during discharge, and the final cutoff voltage.

The capacity of a gelled-electrolyte battery is measured at the 20-hour rate, at 68°F (20°C), to a circuit cutoff voltage of 1.75 volts per cell. The nominal operating voltage is 1.97 volts per cell at this rate. The open-circuit voltage is between 2.10 and 2.20 volts-per-cell, and is relatively unaffected by temperature. Although a discharged cell has a slightly lower open-circuit voltage, its value is not a reliable indication of the state-of-charge.

The gelled-electrolyte battery; like the nickel-cadmium battery; is discharged at the 20-hour, 15-hour, 5-hour, and 1-hour rates. These rates are also designated as C/20, C/15, C/5, and C/1 respectively; with higher rates again designated as 2C, 3C, etc.

Full rated capacity for a gelled-electrolyte battery is normally reached after 10 or more full charge/discharge cycles.



### CAPACITY PERFORMANCE NOMOGRAM

Figure 3-13, illustrates a sample nomogram supplied by some manufacturers. To determine discharge time and available capacity at a specific discharge rate, align a straight edge with the pivot point and the desired discharge rate (Column B). Next read time and capacity at the point where the straight edge intersects (Column A).

**Example:** 4-ampere discharge rate = 6.5 ampere-hours (1.5 hours discharge time).

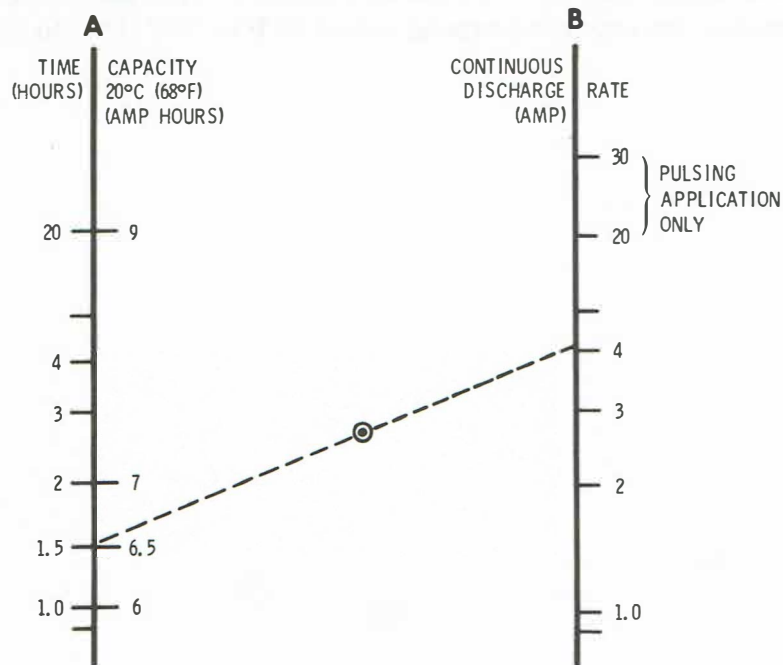


Figure 3-13  
Capacity Performance Nomogram.

## **Gelled-Electrolyte Batteries And Pulse Applications**

The maximum continuous current discharge rate for a gelled-electrolyte battery is normally considered to be 15 amps.

At discharge currents beyond 15 amps, rest periods are required. When discharging current in the 15 to 20 amp range, the maximum "on time" is 2 minutes with a minimum rest period of 8 minutes. In the current discharge range from 20 to 30 amps, the maximum on time is reduced to 1 minute and the minimum rest time is increased to 9 minutes. Under no circumstances should the battery be discharged at pulsed rates exceeding the maximum ratio listed on the nomogram for each battery. These time values assume ambient temperatures of 59°F to 77°F (15°C to 25°C).

## Charging Characteristics and Charging Techniques

### CHARGING PRINCIPLES

To recharge a gelled-electrolyte battery, a DC voltage greater than the open-circuit voltage of the battery under charge must be applied to the terminals (positive terminal of charger to positive terminal of the battery, and negative terminal of the charger to the negative terminal of the battery). This applied voltage overcomes the back electromotive force of the battery and allows the charging current to flow. The amount of current flowing will depend upon a number of factors including the applied voltage (charging voltage) and the state of the charge of the battery. The back electromotive force or "on-charge battery voltage" varies with the state of charge and tends to regulate the amount of current flowing into the battery. Gelled-electrolyte batteries, when placed on charge, approximate the voltage and current characteristics which change with time (state of charge). These characteristics are shown in Figure 3-14.

Actual voltage and current charging curves vary widely depending on the charger being used, the battery, and the temperature. The battery design dictates that this charging current should be maintained within certain limits at different stages of charge.

These values are given in the manufacturer's specification sheets for each battery.

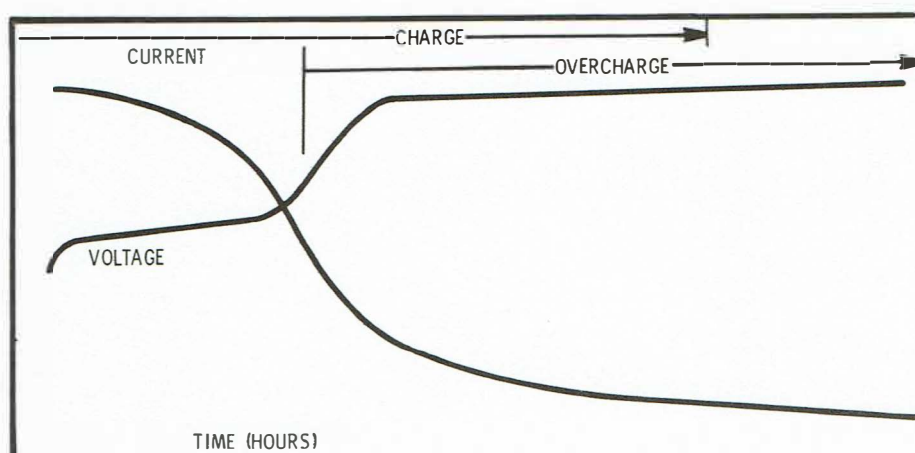


Figure 3-14  
Typical "On-Charge" Voltage and Current Curves.

### CHARGER TYPES

A battery can be charged with any number of different chargers having different characteristics. Selecting a charging system will involve trade-offs between desired battery life, the amount of time available for re-charge and the cost of the charging system, which is influenced by the power output and regulation characteristics. Every application will have different requirements, and it is important to evaluate these factors to obtain the most satisfactory results or compromises.

The life of all gelled-electrolyte batteries is usually determined by the amount of overcharge. It is important to remember that it is not the total accumulated overcharge capacity, but rather the rate of overcharge that greatly affects battery life. Overcharge leads to failure by corroding the positive grid and/or drying out the electrolyte solution. The ideal choice of a charging system, from the standpoint of battery life, is one that prevents high-rate or excessive overcharge while at the same time maintaining a fully-charged battery.

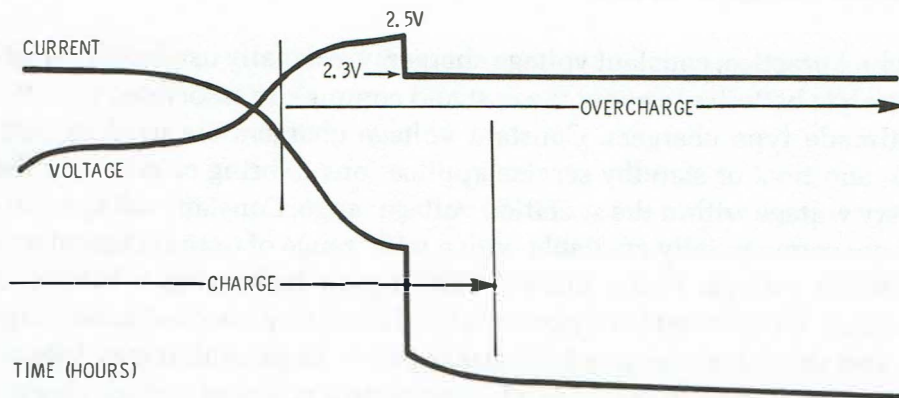


Figure 3-15  
Ideal (Multimode) Charger Voltage and Current Curves.

### Ideal (Multimode) Charger.

The fastest way to fully recharge a battery and still avoid excessive overcharging, is with a charger that has two or more output voltage levels. As an example, a battery can be recharged initially at a current not exceeding 20% of its rated capacity until a specified voltage (counter EMF) is reached. When the specified voltage is reached, the charger automatically switches to a lower output voltage, at which point the current acceptance of the battery is reduced. More complex charging circuits have also been developed that incorporate additional features such as a third output level, output current sensing feedback control and output temperature compensation. Since batteries are made of two or more cells in series, this voltage must be high enough to compensate for the slight capacity variations between cells.

A recommended charging method is illustrated graphically by the current and voltage curves shown in Figure 3-15. In this example, the charger operates initially at a high current output mode until the battery voltage reaches an average value of 2.5 volts/cell. The charger then switches to a 2.3 volts/cell float or maintenance voltage level. A float voltage is selected to provide sufficient current to keep all cells in the battery in a fully-charged condition by overcoming any internal capacity losses.

It should also be noted that while multimode charging will require more components than comparable constant voltage or constant current chargers, solid-state integrated circuits can greatly reduce the cost difference.



### Constant Voltage Chargers

In actual practice, constant voltage chargers are usually used with gelled-electrolyte batteries to avoid the cost and complexity associated with the multimode type chargers. Constant voltage chargers are used in both cycle and float or standby service applications to bring or maintain the battery voltage within the specified voltage range. Constant voltage chargers are commercially available with a wide range of output current and regulation ratings. Initial current limiting on recharging a battery is necessary, but is usually a characteristic of a suitably matched power supply, and should not be an additional expense. In general, it may take almost twice as long to recharge a battery with a constant voltage charger than with a multimode charger, but this disadvantage is more than offset by the lower cost and simpler circuit of the constant voltage charger. The change of on-charge voltage and current with respect to time will be essentially that illustrated in Figure 3-14. Exact values must be measured with the actual charger, battery, and environment of application.

Two major circuits required in a constant voltage charger are the transformer/rectifier and regulator circuits, shown in Figure 3-16. The transformer/rectifier converts the high voltage AC from the power line to low voltage rectified AC which has an average DC equivalent. A full-wave bridge rectifier supplies current during both half cycles of the AC power input and provides a more continuous current than is obtained with half-wave rectification.

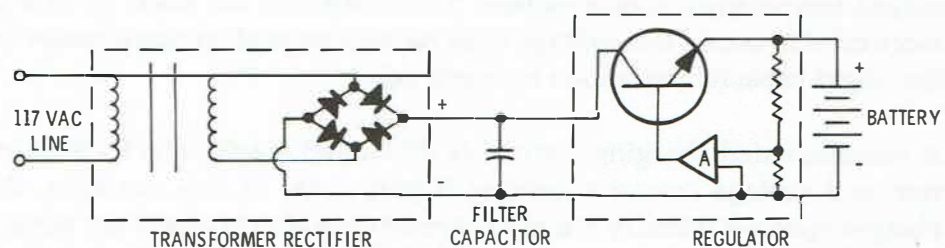


Figure 3-16  
Constant Voltage Charger.

The basic function of the voltage regulator is to maintain a constant voltage at the battery terminals, as the current acceptance of the battery decreases with charge restoration. Such a regulator is connected in a series-parallel relationship with the battery, and drops, or holds in reserve, a varying amount of the DC input voltage to give a regulated output voltage. This regulator must be able to compensate to some extent for variations in the power line voltage and the battery load. Temperature compensation is also included in some units.

The voltage regulator may be regarded as an electronic servomechanism with closed loop control. Feedback signals are compared, inverted, and amplified by control amplifier A and the resulting output difference or error voltage is applied to the base of the pass transistor. In effect, the pass transistor becomes a controlled variable series resistance. Feedback, which is a function of the controlled variable, is determined by the resistive voltage divider and is compared with the reference input of amplifier A. Amplifier A drives, or biases, the pass transistor in such a manner that the error signal applied to amplifier A approaches zero.

The DC output of the transformer/rectifier is usually filtered to reduce or remove the superimposed AC "ripple" by a capacitor in parallel with the output. This filter capacitor, in effect, stores additional charge when the AC ripple component becomes more positive, and releases this charge as the AC component voltage becomes less positive. To be effective, the parallel capacitor must have a relatively small impedance in comparison with the load. The initial charging currents may be excessive for a particular battery if the filter capacitance is too high.

The filter capacitor is sometimes omitted when the constant voltage power supply is to be used only for battery charging. Superimposed AC ripple or pulsed currents that are applied to the battery during recharge or float are not detrimental, provided the currents are not excessively high. However, pulse type chargers, such as those using a silicon controlled rectifier (SCR) where the pulse duty cycle is 50% or less, are not recommended for recharging a battery. Both AC ripple and noise currents in the order of  $\pm 10\%$  of the average DC current level are sometimes present during the initial charging period. These fluctuating currents are normally introduced by the line voltage or the power supply. The AC and random noise decrease and often disappear as the battery becomes fully charged and the current approaches a small, quasi-stable float value.

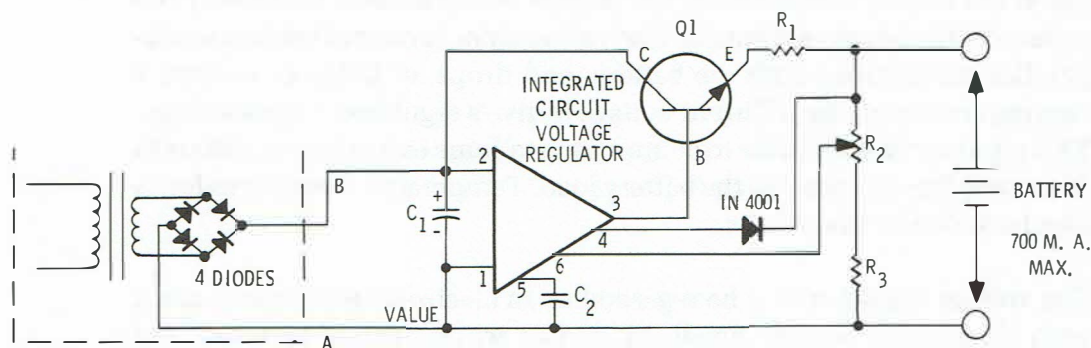


Figure 3-17  
Solid-State, Constant-Voltage Float Charger.

### Float Charger

A typical, solid-state, constant voltage float charger is shown in Figure 3-17. This particular constant voltage charger uses an integrated circuit (IC) as a positive voltage regulator. This circuit includes a current limiting resistor,  $R_1$ , in series with the transistor output which limits the maximum battery charging current to 0.7 amps. The maximum current can be increased or decreased by varying the resistor value. The output voltage, for example, 2.3 volts per cell, is determined by the setting of potentiometer  $R_2$ . Resistors may be substituted for the 2.5 k potentiometer once the proper value output value has been determined. Output voltage can be set initially by using a 1 k ohm, 1-watt load resistor in place of the battery. The series pass transistor,  $Q_1$ , must be adequately heat-sinked for a maximum temperature rise above ambient of 150°F (65°C).

### Constant Current Charger

With the constant current or trickle charge method, the charging current is held steady, and the voltage varies with time, depending on the state of charge. This method is not recommended because of the long charge time required and/or excessive overcharge resulting from the high float current phase of the charging. This is illustrated by the fact that, at room temperature, the float current is often in the  $C/100$  to  $C/400$  range, where  $C$  is the rated ampere-hour capacity of the battery. It would be impractical to select a current that will both recharge and float a battery.

## Charging Considerations

### INITIAL CHARGE CURRENT

Each battery design will dictate a maximum current that should not be exceeded during charging. For most gelled-electrolyte batteries, the maximum charging current is recommended to be no more than 3 to 4 times the 20-hour discharge current for that particular battery. For example, a 5.0AH battery has a 20-hour discharge rate of 250 milliamperes; therefore, the maximum recommended charge current is between 750 milliamps and 1.0 amp. This maximum range applies to the initial charge current, and current should be reduced as the battery approaches a fully-charged state.

### END OF CHARGE CURRENT

Fully-charged gelled-electrolyte batteries on float or maintenance charge will normally accept small charging currents within a specific range. This range depends on terminal voltage, temperature, and the ampere-hour capacity of the battery. At room temperature, a 6-volt battery floated at 7 volts will usually accept continuously from 1 to 3 mA per ampere-hour of rated capacity. To give a more specific example, the Elpower EP645C gelled-electrolyte battery is rated at 4.5 AH at the 20-hour discharge rate, and normally accepts a float current of from 5 to 10 mA at 70°F (21°C).

### TEMPERATURE EFFECTS

If the in-service battery temperature is greatly different from room temperature, or if large temperature variations are encountered during operation, the charging voltage must be compensated for temperature to realize maximum battery performance and life. Two recommended charging voltage ranges as a function of temperature are shown in Figure 3-18. The higher voltage rating #1 is recommended where a charger with one output float voltage is used both for charging and maintaining the charge. If the battery is charged initially at a higher voltage, it may then be floated in voltage range #2, shown in the shaded area, to maintain it in a fully-charged condition.

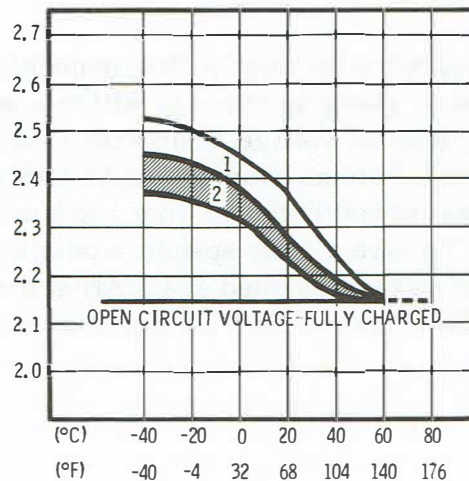


Figure 3-18  
Voltage vs Temperature.



## General Information

The following recommendations are made for the use and care of gelled-electrolyte batteries for trouble-free performance and longer life.

### CIRCUIT CONNECTION

Like the nickel-cadmium battery, the gelled-electrolyte battery requires welded, soldered, or crimped connections for medium and high discharge applications. Again, if pressure connections are required, take care to ensure a firm contact on a clean surface.

### SERIES AND PARALLEL USE

Unlike the nickel-cadmium battery, which can only be used in series, the gelled-electrolyte battery can be used in series, parallel, and series-parallel configurations. Each battery however, should have the same rated capacity and be cycled at least once before it is used. This cycling will equalize the minor capacity differences; thus reducing possible circulating currents between batteries.

### CHARGING

The gelled-electrolyte battery should be charged as soon as possible after it is discharged and before it is stored. Leaving the battery in a discharged condition for prolonged periods causes sulfation on the positive and negative plates. This will either reduce battery capacity, or eventually cause battery failure. **CAUTION: ALL LEAD-ACID BATTERIES, AND GELLED-ELECTROLYTE BATTERIES, RELEASE HYDROGEN GAS TO SOME EXTENT, PARTICULARLY WHEN BEING CHARGED. HYDROGEN GAS WITH OXYGEN IS EXPLOSIVE AND CAN CAUSE SERIOUS INJURY. DO NOT CHARGE GELLED-ELECTROLYTE BATTERIES IN ENCLOSED AREAS. INSURE ADEQUATE VENTILATION TO PREVENT BUILD-UP OF EXPLOSIVE GASES. KEEP SPARKS, FLAME, AND CIGARETTES AWAY.**

## **BATTERY HANDLING**

When you handle gelled-electrolyte batteries, use the same care as with nickel-cadmium batteries. Also, you should operate them in the same cool environment as nickel-cadmium batteries.

## **BATTERY FAILURES**

Gelled-electrolyte batteries are also susceptible to both reversible and permanent failures.

### **Reversible Failure**

Reversible failures are usually due to long storage or high temperature operation, and the battery appears to have lost capacity. In this case, you can usually restore the battery by putting it through 3 to 5 charge/discharge cycles.

### **Permanent Failures**

Permanent failures are generally caused by an internal short within a cell, or by an open circuit within the battery.

Internal shorts can be either high resistance or low resistance shorts, which prevent the battery from accepting a full charge, or delivering full capacity after charging. This is normally caused by the eventual deterioration of the active material in the battery, and is the primary cause of failure.

Open-circuit failures come from premature dry-outs of the cell, and loss of electrolyte, thereby stopping the ability of the cell to conduct electricity between the electrodes. This loss of electrolyte is caused by high temperatures, high drain rates, and high charge rates, which produce venting of one or more cells. A gelled-electrolyte battery is normally considered to have reached the end of life when the battery capacity, after proper charging, yields less than 50% of specified capacity.

## Nickel-Cadmium vs Gelled-Electrolyte

Nickel-cadmium batteries are better for light weight, longer life, a high number of recharge cycles, or rapid charge requirements. Compared with the nickel-cadmium battery the gelled-electrolyte battery is a high energy, low-initial cost system. It is best suited to operations where high energy requirements, standby power, and moderate cycle life is required.

## Programmed Review

13. Gelled-electrolyte batteries are \_\_\_\_\_ expensive than nickel-cadmium batteries.  
(more/less)

14. (less) In a gelled-electrolyte battery, the safety relief valve provides safe operation in the event of \_\_\_\_\_ build-up within the cell.

15. (gas or pressure) Storage or shelf life of a gelled-electrolyte battery will be \_\_\_\_\_ if it is stored at higher temperatures.  
(increased/reduced)

16. (reduced) The gelled-electrolyte battery should be discharged at \_\_\_\_\_ rates to prolong life.  
(high/low)

17. (high) A rest period may be required when a gelled-electrolyte battery is used for \_\_\_\_\_ operation.

18. (pulse) When recharging a gelled-electrolyte battery, a DC voltage greater than the \_\_\_\_\_ voltage of the battery under charge must be applied.

19. (open-circuit) For charging a gelled-electrolyte battery, a charger with two or more output \_\_\_\_\_ levels is preferred.

20. (voltage) A gelled-electrolyte battery should be charged by applying a constant \_\_\_\_\_ to the battery under charge.

21. (voltage) When gelled-electrolyte batteries are used in parallel or series-parallel operations, they should be of the same rated \_\_\_\_\_, and be cycled at least once before they are used.

22. (capacity) The release of \_\_\_\_\_ gasses from a gelled-electrolyte or lead-acid battery is very explosive.

23. (hydrogen). The gelled-electrolyte battery is able to withstand \_\_\_\_\_ charge-discharge cycles than the nickel-cadmium  
(more/fewer)  
battery.

(fewer)



## DC MOTORS

Direct-current motors are widely used in robotic applications for two major reasons. (1) Their speed-torque relationship can be varied, providing a wide range of applications. (2) They are readily adaptable to a variety of control circuits, especially simple logic and microprocessor control.

### Characteristics of DC Motors

Before we proceed with the discussion on the operation and control of fractional horsepower (fhp), battery-operated DC motors, we will present a short refresher on some of the basic characteristics of DC motors.

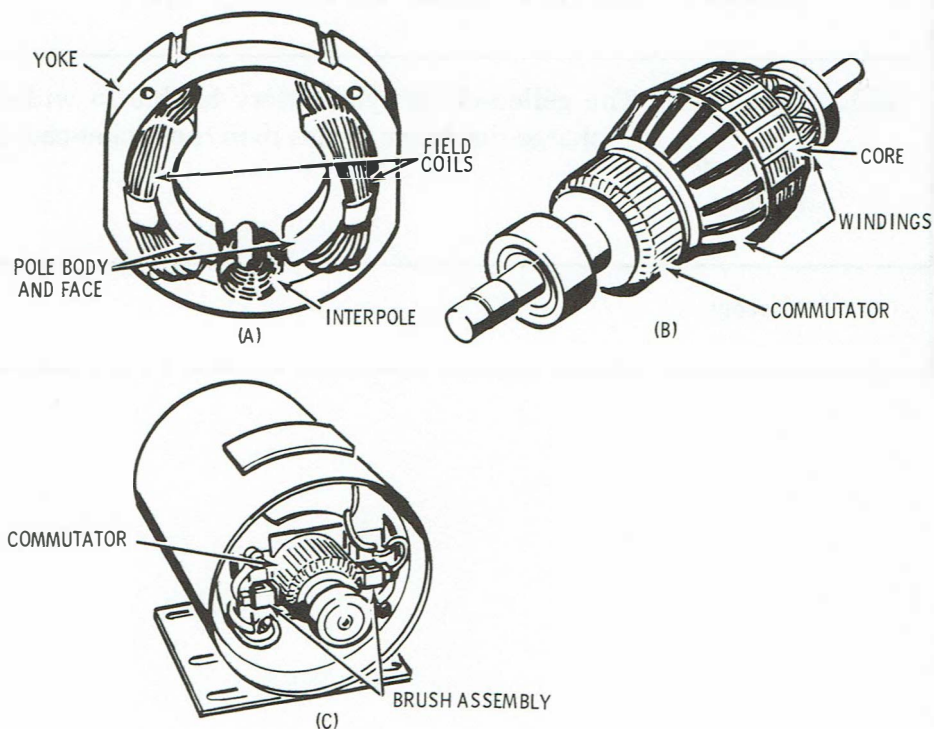


Figure 3-19  
Basic Components of A Wound-Field DC Motor.

## COMPONENTS OF A DC MOTOR

Figure 3-19 shows the basic components of a wound-field DC motor. The **field structure** shown in Figure 3-19A, is used to provide a magnetic circuit for the flux and to hold the field coils. The field structure consists of a yoke, pole bodies, and pole faces. In addition, an inerpole(s) — a small auxiliary pole placed between the main poles to reduce sparking at the commutator — is sometimes included in the field assembly. In some motors, only the pole bodies and pole faces are laminated and bolted to a solid yoke; while in others, the entire field structure may be laminated.

The **field coils** may consist of shunt coils wound with many turns of small wire, connected in series with each other, and placed across the supply, or in parallel with the armature. The field coils can also be series-wound coils, with fewer turns of larger wire, which are connected in series with the armature. In the case of a compound motor, the field coils may be both shunt and series coils. The field coils are wound on the pole body and face assembly to strengthen their magnetic field.

The **armature** assembly consisting of the commutator, the windings, and the core is shown in Figure 3-19B. The armature assembly is formed by winding many lengths of insulated copper wire on a slotted iron core; in turn, each armature winding is connected to a commutator segment. The commutator segments are separated from each other by a small gap, which is filled with an insulating material, such as mica.

The **commutator** and **brush** assembly, shown in Figure 3-19C, is necessary to provide a current path from one side of the supply through the armature windings to the other side of the supply. The commutator and brushes, together with the armature windings, are so arranged that current passing from one brush to the other flows through half of the armature windings in one direction and through the other half in the opposite direction. The direction of current flow causes a north pole to be generated in one-half of the armature windings and a south pole to be generated in the other half. The resulting armature magnetic fields attract and repel the magnetic fields produced by the corresponding field coils, resulting in armature rotation. When the brushes come into contact with the next segment of the commutator, current still flows in the same direction; but this time, the armature windings produce magnetic fields of the opposite polarity, causing further armature rotation.

## TORQUE

Most AC motors will stall at torque loads above twice their rating, and cannot start loads much above 150% of their rated torque. On the other hand, DC motors are capable of delivering three or more times their rated torque for short periods of time. Furthermore with the proper power source, DC motors can provide over five times their rated torque without stalling, in emergency situations.

## SPEED REGULATION

Speed regulation is the ability of a motor to maintain its speed when a load is applied. It is an inherent characteristic of a motor and remains the same as long as the applied voltage does not vary. The speed regulation of a motor is a comparison of its no-load speed to its full-load speed and is expressed as a percentage of full load-speed. Thus,

$$\text{percent speed regulation} = \frac{\text{no-load speed minus full-load speed}}{\text{full-load speed}} \times 100$$

For example, if the no-load speed of a DC motor is 1800 rpm and the full-load speed is 1700 rpm, the speed regulation is

$$\frac{1800-1700}{1700} \times 100 = 5.9\%$$

The **lower** the speed-regulation percentage figure of a motor, the more constant the speed will be under varying load conditions and the **better** will be the speed regulation. Conversely, the **higher** the speed-regulation percentage figure, the **poorer** is the speed regulation.

## **SPEED VERSUS TORQUE**

The speed at which a DC motor rotates depends upon the strength of the magnetic field between the stator and the rotor. As field strength increases, so does speed of rotation. Torque or turning force of a DC motor is proportional to the current in the rotor coils, called the armature, and the strength of the magnetic field in the stator coils, called the field coils. Hence, for a given motor load (torque) the speed of rotation can be controlled by varying the current in the field windings.

## **TYPES OF DC MOTORS**

The three basic DC motor types are the shunt-wound, series-wound, and compound-wound. These names refer to the manner in which the field coils are connected in reference to the armature.

## Shunt-Wound

The shunt-wound motor, shown in Figure 3-20A, has a high resistance field coil (many turns of small wire) connected in parallel across the armature circuit. The variable resistor in series with the field coil provides a means of varying the field strength, and thereby the motor speed. This type of motor, with a constant voltage applied, develops variable torque at an essentially constant speed, even under changing load conditions. Most shunt-wound motors are operated from adjustable supplies.

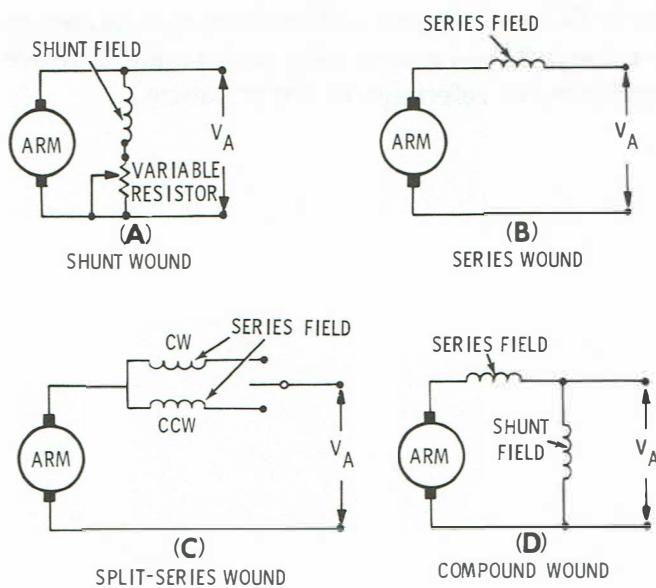


Figure 3-20

Types of DC Motors:

- (A) Shunt-Wound.
- (B) Series-Wound.
- (C) Split-Series Wound.
- (D) Compound-Wound.



### Series-Wound

A series-wound motor, shown in Figure 3-20B, has a low resistance field coil (few turns of a large wire) connected in series with the armature circuit. Since the same current flows through both the armature and the field winding, field strength and armature current are both high at start-up. This results in a high starting torque, since torque is proportional to field strength and armature current. In the series-wound motor, speed will vary greatly because the field strength changes with armature current, which in turn changes with the load. In other words, the speed is low under heavy loads, but becomes excessively high under light loads. These type motors must not be operated without a load, because they can speed up enough to damage themselves. This rapid acceleration under a no-load condition is referred to as “run away.”

Another version of the series motor, the split series motor, seen in Figure 3-20C, has two oppositely wound field coils. This type of motor is easy to reverse by switching the applied voltage from one field coil to the other.

### Compound-Wound

The compound-wound motor, shown in Figure 3-20D, has one set of weak field coils in series with the armature circuit and another set of strong field coils in parallel with the armature circuit. Thus, the compound-wound motor is a compromise between the shunt motor and the series motor. It develops a greater starting torque than the shunt motor and, at the same time, provides less speed variation than the series motor.

The speed regulation parameter of a compound motor is normally specified at the full-load speed and at one other point, usually at no-load speed.

## SPEED CONTROL OF WOUND-FIELD DC MOTORS

There are two basic methods of controlling the speed of a wound-field motor; shunt-field control, and armature-voltage control.

### Shunt-Field Control

A tape deck reel drive motor is a good example of an application in which speed control is critical. In tape drives, the tape must be wound on the reel at a constant linear speed and constant tension, regardless of reel diameter.

This control is obtained by weakening the shunt-field current of the motor to increase speed and to decrease output torque for a given armature current. Since the rating of a DC motor is determined by heating, the maximum allowable armature current is nearly constant over the speed range. Hence, at rated current, output torque varies inversely with speed; thus, the motor has constant horsepower capability over its speed range.

This type of control is only suitable for obtaining speeds greater than the base or slowest operating speed. A momentary speed reduction, below base speed, can be obtained by overexciting the field, but prolonged overexcitation will overheat the motor. In addition, magnetic saturation in the motor permits only a slight reduction in speed for a large increase in field voltage.

The maximum speed range using field control is usually considered to be 3:1; and this occurs only at low speeds. Some special motors have greater speed ranges, but if the speed range is much greater than 3:1, some other control method is used for at least a portion of the range.

### Armature-Voltage Control

In armature-voltage control, shunt-field current is maintained at a constant level, from a separate source, while voltage applied to the armature is varied. The speed is proportional to the counter-emf (opposing voltage in an inductive circuit), which is equal to the applied voltage minus the armature-circuit IR drop. At rated current, the torque remains constant regardless of the speed (since magnetic flux is constant) and, therefore, the motor has constant torque over its speed range.

One major parameter, heat, must not be overlooked when using armature-voltage control, especially in a self-ventilated motor. As motor speed is lowered, the ventilation capability of the motor is also reduced; therefore, the motor cannot be loaded with quite as much armature current without exceeding the rated temperature rise.

## Motor Selection Factors

When selecting a DC motor for a given application, several factors must be taken into consideration. The following is a synopsis of some of the more important factors to be considered.

### SPEED RANGE

In applications requiring a large speed range, and where field-control is used, the base speed must be proportionately lower and the motor size must be increased. For speed ranges over 3:1, armature-voltage control should be considered for at least a portion of the range. Unlimited speed range can be obtained using armature-voltage control; however, below approximately 60% of base speed, the motor rating should be decreased, or the motor should only be used for short periods of time.

### SPEED CONTROL UNDER LOAD

When a specific application calls for constant speed under varying load conditions, a shunt-wound motor should be used. If motor speed regulation must be kept to an absolute minimum, a speed regulation circuit employing tachometer feedback may be used. If the application requires a decrease in motor speed when the load increases, a series-wound or compound motor may be used.

## Reversing a DC Motor

DC motors may be reversed by reversing the polarity of the voltage applied to the armature circuit, relative to the field winding. The switching usually takes place in the armature because it has less inductance and, therefore, will cause less arcing at the switch contacts. This operation affects the power supply and control circuit, and may also affect the brush adjustment of the motor, if the motor cannot be stopped before switching to reverse operation. If this is the case, a suitable armature-voltage control system should be used.

## Programmed Review

24.	A small pole, known as an _____, is sometimes placed between the main poles in a DC motor to reduce commutator sparking.
25.	(interpole) The brush assembly provides a current path from the supply through the _____ to the armature windings.
26.	(commutator) DC motors are capable of supplying _____ torque than AC motors. (more/less)
27.	(more) In a DC motor, the higher the speed regulation percentage, the _____ is the speed regulation. (better/poorer)
28.	(poorer) In the _____ DC motor, speed will vary greatly with an increase or decrease in the load.
29.	(series) The split series DC motor has two field coils wound in the _____ direction for easy reversal of the motor. (same/opposite)
30.	(opposite) When using armature-voltage speed control, the temperature of the motor will _____ as motor speed decreases. (increase/decrease)
31.	(increase) When reversing a DC motor, switching usually takes place in the _____ circuit because it has less inductance.
	(armature)

## DC BRUSHLESS MOTORS

The DC motors previously discussed all have one common undesirable characteristic; they use a brush-commutator assembly to mechanically switch the armature current. Because of this mechanical switching action, the brushes have a limited life and cause commutator wear. In addition, brush wear produces brush dust, which in turn can foul the motor bearings and also create a voltage leakage path. In a continuous rotating brushless motor, mechanical switching (commutator action) is replaced with electronic switching. Brushless motors are not simply AC motors powered by an inverter, but have position feedback so that the input waveforms are kept in proper timing with respect to rotor position.

The continuous rotating brushless motor is very versatile. For instance, it accelerates from zero to operating speed as a permanent magnet DC motor; and once it has reached operating speed, it can be switched to synchronous operation or operated in a phase-locked loop.



## Transistor Switched Brushless DC Motor

Some brushless motors use optics and photocells to commutate the motor current, but the transistor switched motor shown in Figure 3-21 is the most common. The motor itself is a single-phase permanent-split capacitor motor with a center-tapped main winding. The switching circuit is a simple push-pull oscillator, which fires alternately through each leg of the main winding. The switching circuit could be classified as a basic astable (free-running) multivibrator; however, in an astable multivibrator, current and voltage change abruptly from one value to another and the frequency is determined mainly by the values of  $R$  and  $C$  in the circuit. In this oscillator, the oscillation is controlled by a tank circuit (in this case, the main winding), and there is no abrupt change; therefore, the transition from one state to another is sinusoidally smooth.

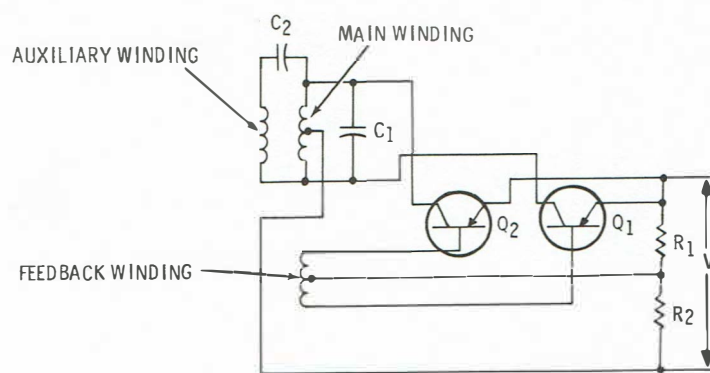


Figure 3-21  
Transistor-Switched DC Brushless Motor.

Like all oscillators, this circuit requires a feedback. In this case, regenerative (positive) feedback supplied by a winding wound in the stator slots to generate a control voltage which determines the operating frequency. Biasing resistors (R1 and R2) establish the proper biasing level of the switches (transistors Q1 and Q2) and also help start the motor. A commutating capacitor (C1) is placed across the main winding to reduce voltage peaks and keep the resonant frequency of the circuit at the operating frequency. The auxiliary winding and capacitor (C2) aid in starting the motor, using the same principle as the split-phase capacitor induction motor, discussed in Unit Two. In fact, the operation of the transistor-switched brushless DC motor is very similar to the AC single-phase capacitor-run motor.

A disadvantage of this motor is its inability to develop a large starting torque. As a result, it is suitable for driving only very low-torque loads such as blowers. Of course, gearing could be used to increase the torque, but at the expense of a decrease in speed. When used in a low voltage system, the brushless motor is not very efficient. The switching circuit requires at least a 1.5-volt drop which, in a 12-volt system, reduces the comparative efficiency by over 20%. Also, since only half of the main winding is in use at any instant, copper losses, ( $I^2R$  loss) are quite high.

Advantages, however, greatly outweigh these disadvantages in certain applications. Since there are no brushes and commutator, motor life is limited only by bearing lubrication. Also, since mechanical switching is replaced by electronic switching there is no contact arcing. Thus, electrical noise and explosion hazards are also reduced.

## Hall-Effect Motor

The Hall-effect motor is another type of DC brushless motor that is gaining popularity. This type of motor is noted for its high efficiency, long life, high reliability, low noise, and low power consumption. There are a number of brushless DC motors up to 1/40 hp that use Hall-effect generators for control purposes. Before discussing a specific Hall-effect motor, we will examine the basic Hall-effect principle.

Hall-effect occurs as current flows through a semiconductor material, usually indium or antimonide, that is in the presence of a magnetic field. As electrons or holes flow through the semiconductor material, they experience a force whose direction depends on the charge and velocity of the majority carriers (either electrons or holes) as well as the polarity of the magnetic flux field.

For example, assume that the semiconductor device, shown in Figure 3-22A, is a p-type material (majority carriers are holes). If control current ( $I_C$ ) flows through the semiconductor material from left to right, and the polarity of the magnetic flux field ( $B$ ) were such as to present a north pole to the semiconductor material; the force ( $F$ ) is applied downward on the majority carriers. This action causes the majority carriers to collect on the bottom surface of the semiconductor material; thus generating a positive Hall-output voltage at the bottom of the semiconductor.

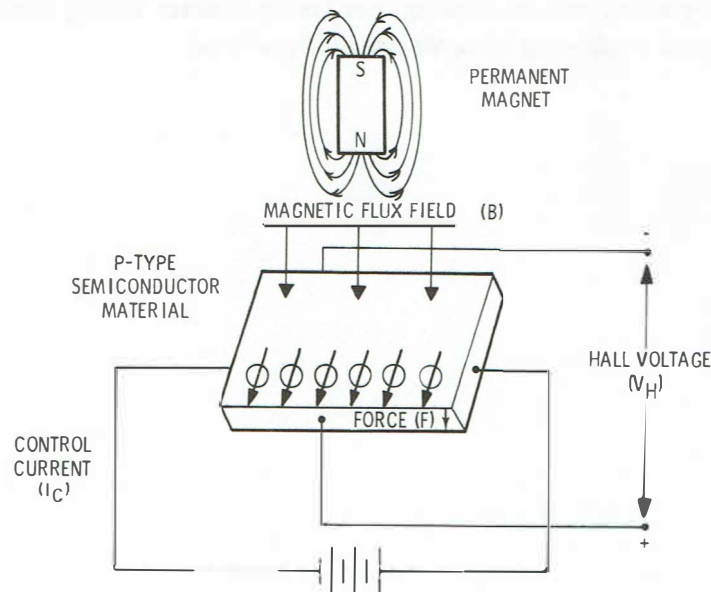


Figure 3-22A  
Basic Hall-Effect Generator

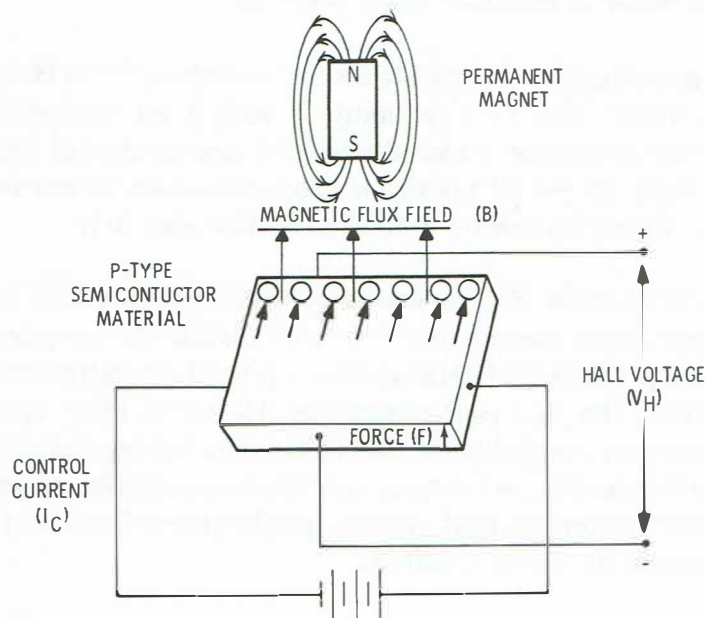


Figure 3-22B

Conversely, if the same device were now presented a magnetic flux field whose polarity is a south pole, as shown in Figure 3-22B, an upward force would be applied on the majority carriers. This action would cause the majority carriers (holes) to collect at the top surface of the semiconductor material. This in turn would cause an excess of electrons at the bottom of the semiconductor, which would produce a negative Hall-output voltage on the bottom surface.

If the semiconductor were an n-type material (whose majority carriers are electrons), the polarity of the Hall-voltage becomes negative on the bottom surface, with a downward magnetic force applied, with respect to the top. Conversely, an upward magnetic force would produce a positive Hall-voltage on the bottom of the semiconductor material.

Precise diffusion of a specific impurity into silicon determines the majority carriers' mobility and the charge density. Therefore, the Hall-effect ideally produces a linear and repeatable Hall-voltage that is proportional to the strength of the external magnetic field.

In addition to magnetic field intensity, other semiconductor material related factors, such as temperature, mechanical stress and current, govern the Hall-voltage. An increase in either mechanical stress (physical pressure) or temperature will affect the majority carriers' mobility; while varying current flow will cause nonlinear fluctuations in the Hall-voltage. A constant current source eliminates nonlinearity, and a temperature-compensating network can offset the thermal effect, but the nonlinearity caused by mechanical stress cannot be corrected.

## OPERATION OF A HALL-EFFECT MOTOR

The **Hall generator** is responsible for the operation of the Hall-effect DC brushless motor. The Hall generator is simply an encapsulated, thin wafer of semiconductor material used for measuring the strength of a magnetic field. Its output voltage is proportional to the current passing through it, times the magnetic field perpendicular to it.

The application of the Hall generator is shown in Figure 3-23. In this case, a 2-pole permanent magnet rotor is placed inside a housing containing a 4-pole stator (P1 through P4) in which are placed the stator windings (W1 through W4). The two Hall generators (HG1 and HG2) are of p-type material and are mounted in the stator assembly, 90 degrees apart. Each of the DC amplifiers (A1 and A2) are actually two amplifiers that sense the polarity of the generated Hall-voltage, amplify the voltage, and switch on the corresponding stator windings.

With the permanent magnet rotor in the position shown, a downward force, created by the rotor's north pole, would be felt on the surface of Hall generator HG1, creating a positive Hall-voltage. Conversely, because of the rotor's position, no Hall-voltage would be produced by Hall generator HG2 at this time. The positive Hall-voltage produced by HG1 is amplified by DC amplifier A1, which in turn, allows current to flow through the appropriate stator winding — in this case, W3.

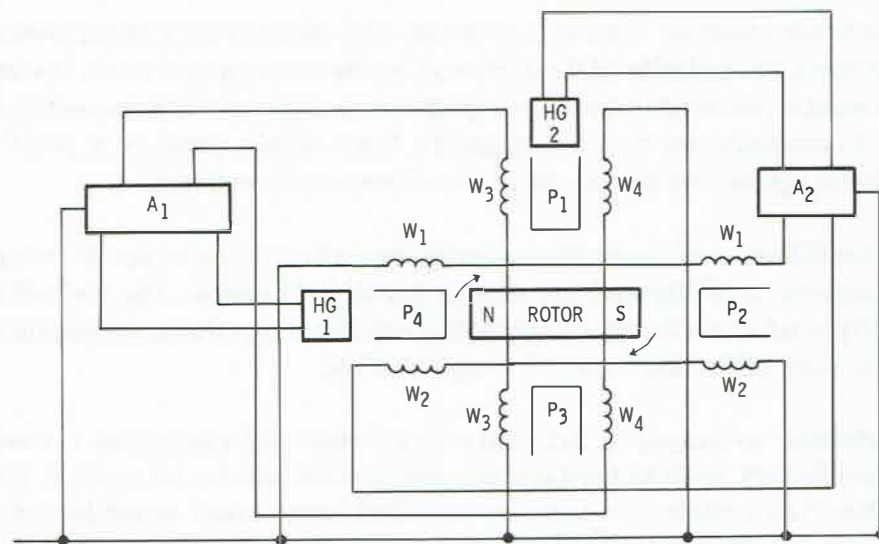


Figure 3-23  
DC Brushless Motor Using Hall Generators.



Stator winding W3 is so wound as to produce a south pole on P1 and a north pole on P3 when current flows through it. This will cause the rotor to rotate  $90^\circ$  in the clockwise direction. At this instant, the rotor's north pole now creates a downward force on the surface of HG2, producing a positive Hall-voltage to be generated by HG2. Consequently, because the rotor has moved  $90^\circ$ , no force is felt on HG1; thus, there is no Hall-voltage generated by HG1.

The positive Hall-voltage generated by HG2 is amplified by DC amplifier A2, and in this case, current is allowed to flow through stator winding W2. Again, because of the way the stator poles are wound, current flow through stator winding W2 causes P2 to become a south pole and P4 to become a north pole. The result is an additional  $90^\circ$  of clockwise rotation.

Since the rotor has completed  $180^\circ$  of rotation, Hall generator HG1 now senses the rotor's south pole. You know from the previous discussion on Hall-effect, that a south pole will cause the P-type semiconductor to feel an upward force, which in turn, will cause a negative Hall-voltage to be generated. This negative Hall-voltage is amplified by the other section of DC amplifier A1, which in turn, allows current to flow through stator winding W4. Current flows through stator winding W4 in such a manner as to produce a north pole at P1 and a south pole at P3. Thus, we have an additional  $90^\circ$  of rotor rotation, for a total of  $270^\circ$ .

Hall generator HG2, now sensing a south rotor pole, produces a negative Hall-voltage, which in turn, is amplified by the other section of DC amplifier A2. Amplifier A2 now permits current to flow through stator winding W1, which produces a north pole at P2 and a south pole at P4. This creates another  $90^\circ$  of rotor rotation, or one complete revolution. As a result of the previous action, a rotating magnetic field has been produced by the stator coils in the sequence W3, W2, W4, and W1.

If the Hall generators were supplied by a constant current source, the rotor would turn at a constant speed. Rotation speed can be changed by varying the amount of current through the Hall generators, which in turn, affects Hall-voltage output. An increase or a decrease in Hall-voltage will cause the DC Amplifiers to conduct sooner or later, due to preset amplifier bias. This results in a faster or slower switching of the stator windings, which in turn, effects speed of rotation of the motor. One method of maintaining a constant speed would be to sense "back emf" of the motor and compare it to a reference voltage representing desired speed. These two voltages could be fed to a difference amplifier whose output could be used to control the current through the Hall generators.

## Programmed Review

32. In a continuous rotating brushless DC motor, \_\_\_\_\_ switching is replaced by electronic switching.

33. (mechanical) A continuous rotating brushless DC motor \_\_\_\_\_ be switched to synchronous operation, once it has  
(can/cannot) been brought up to operating speed.

34. (can) A transistor switched brushless motor requires a \_\_\_\_\_ circuit to determine the operating frequency.

35. (feedback) The transistor switched brushless motor \_\_\_\_\_ very efficient in a low voltage system. (is/is not)

36. (is not) The output voltage of a Hall-effect generator is proportional to the \_\_\_\_\_ passing through it, times the mag-  
(current/voltage) netic field perpendicular to it.

37. (current) Speed of rotation of a Hall-effect motor can be accomplished by varying the amount of \_\_\_\_\_ through the Hall generators.

(current)

## STEPPER MOTORS

Stepper motors are becoming increasingly popular in industrial applications where computer control is the norm. They offer significant advantages over the usual closed-loop servomotor systems found in many industrial control settings. With steppers, feedback signals are not required, and motor error is noncumulative as long as pulse-to-step integrity is maintained. For example, a train of pulses can be counted into a stepper, and its resultant final position will be known within a small percentage of one step.

### Stepping Motor Defined

A stepping motor is a motor that has the capability of rotating in either direction, starting or stopping at various mechanical positions, and moving its rotor in precise angular increments for each input excitation change or step. The precise angular movement is repeated for each input step command, which results in the motor's ability to accurately position its rotor in a known repeatable direction.

Rotor position, speed, direction, and distance of angular travel are easy to control in a stepper motor. Since each input step moves the rotor to a known position, the only rotor error is the single step accuracy of the motor, regardless of direction or distance of travel. In some stepper motors, this error is less than 1% of one step. The number of steps required to complete one revolution of the rotor varies, depending on the specific motor and its intended application.

Stepper motors are typically available in the steps-per-revolution sizes shown in Figure 3-24. Each stepper motor is built for a specific step angle; however, they may be operated at one-half the given step angle, but at reduced torque.

	INCREMENTAL ROTOR ANGLE PER STEP
240	1.5°
180	2.0°
144	2.5°
72	5.0°
48	7.5°
24	15°
12	30°

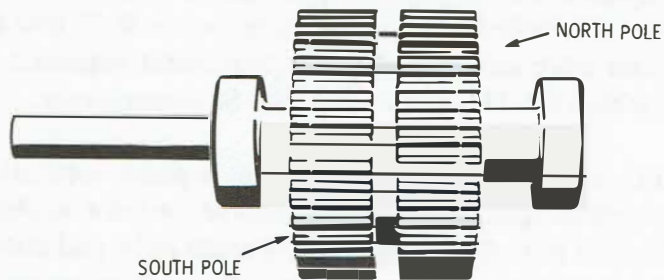
Figure 3-24  
Typical Stepper Motor Sizes.

## Stepper Motor Types

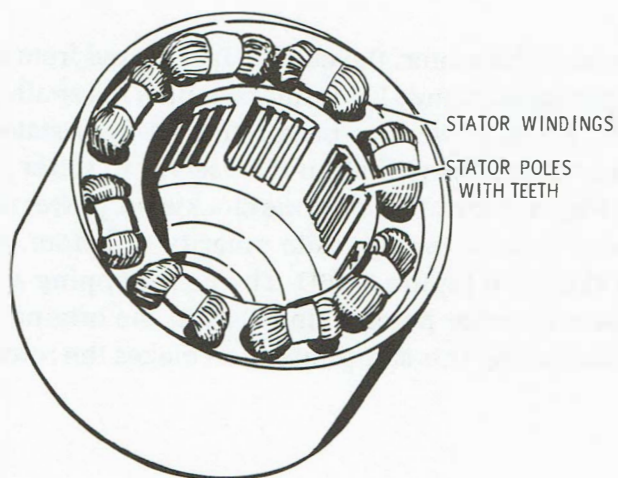
There are numerous types of stepper motors; but they can all be grouped into three major classifications: permanent magnet, variable reluctance, and bifilar wound. We will examine the basic characteristics and operation of each of these three major types of steppers.

### BIPOLAR PERMANENT MAGNET STEPPER

The bipolar permanent magnet stepper contains a permanent magnet rotor whose operation is a result of magnetic characteristics: like poles repel and unlike poles attract. The permanent magnet rotor, shown in Figure 3-25A, is an axially-oriented permanent magnet with a gear-like hub on each end of the magnet. The north pole has teeth that are  $180^\circ$  out of phase from the south pole. The stator poles, shown in Figure 3-25B, also have teeth; but in this case, the magnetic poles are generated by the stator windings. **Note that the number of teeth on the rotor is different from that of the stator.** This is necessary so that the teeth of the rotor will never be lined up exactly with the teeth of the stator. It is this characteristic that actually determines the predictable movement of the rotor, since there is a magnetic attraction between the closest rotor and stator tooth. It should also be noted that the number of teeth, on the rotor and in the stator, will determine the amount of angular movement of the rotor, each time the motor is stepped. The greater the number of teeth, the smaller the step angle. In addition, through the magnetic attraction between the permanent magnet rotor and the stator, the permanent magnet motor is able to provide some low-torque holding power, even when depowered. This characteristic is referred to as residual torque.



(A)



(B)

Figure 3-25

Permanent Magnet Motor:

- (A) Axially-Oriented Permanent Magnet Rotor.
- (B) Stator.

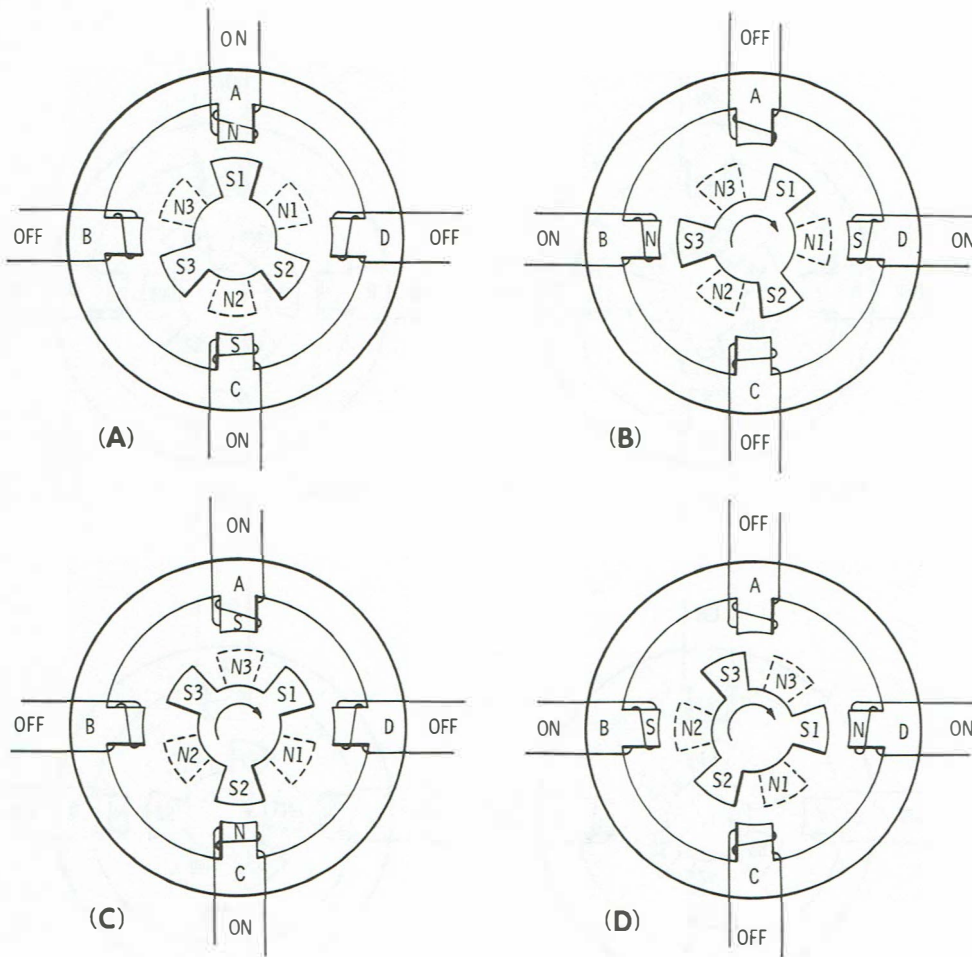


### **Permanent Magnet Stepper Motor Operation**

For ease of explanation, the permanent magnet stepper, shown in Figure 3-26, has four stator windings and poles, labeled A, B, C, and D; and, the axially-oriented rotor consists of three permanent magnetic north and south poles, labeled N1, N2, N3, and S1, S2, S3 respectively.

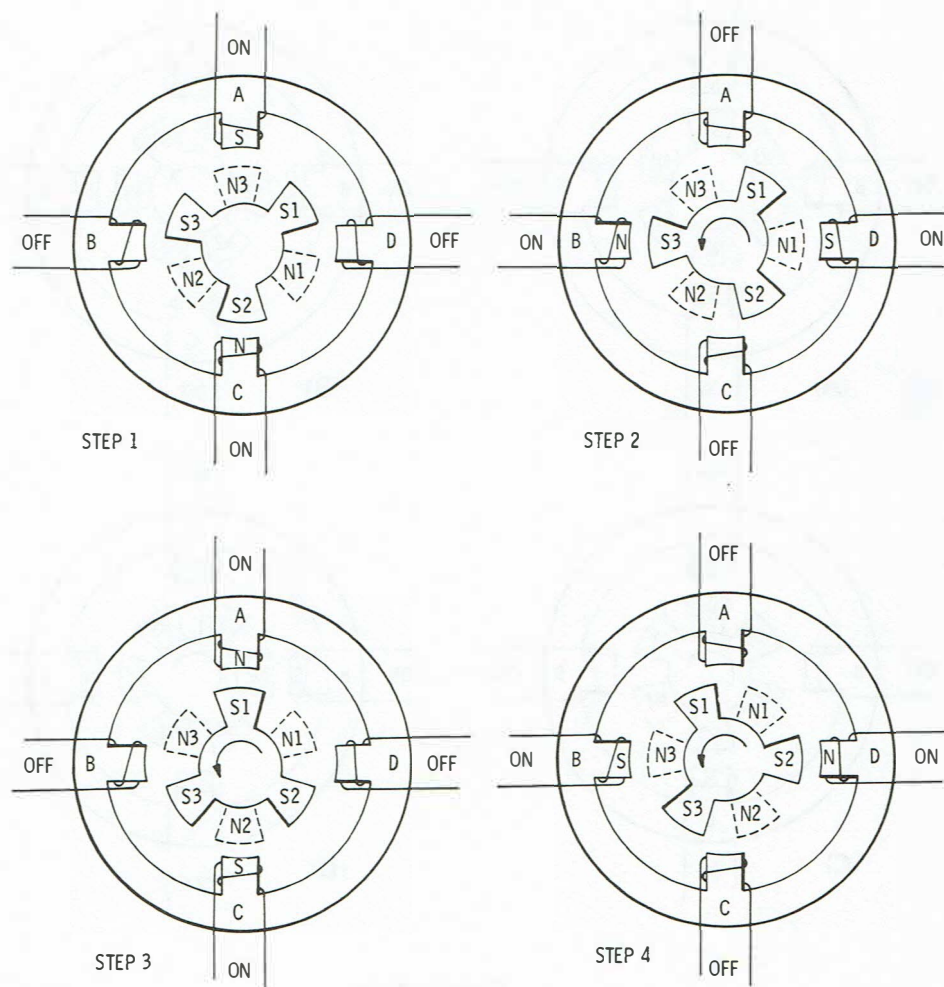
The excitation sequence (in this example, 4-phase excitation) of the windings is fairly simple. Consider the poles of the rotor, as shown in Figure 3-26A; if stator pole A is energized as a north pole, and stator pole C is energized as a south pole (with stator poles B and D not energized), rotor pole S1 will align itself with stator pole A, and rotor pole N2 will align itself with stator pole C. If stator pole B is then energized as a north pole and stator pole D energized as a south pole (with stator poles A and C deenergized), the rotor will rotate clockwise with rotor poles S3 and N1 aligning themselves to stator poles B and D respectively, as shown in Figure 3-26B.

To further step the motor, the current is reversed from its original direction in stator poles A and C, now making pole A a south pole and pole C a north pole. Since power has been removed from stator poles B and D, rotor poles S2 and N3 will align themselves to stator poles A and C, as shown in Figure 3-26C. To continue clockwise, power is applied to stator poles B and D with the opposite polarity of before, resulting in rotor action as shown in Figure 3-26D. The next stepping action would be to apply power to stator poles A and C as in the original position (Figure 3-26A). Continuing this stepping action makes the rotor turn clockwise.



**Figure 3-26**  
Bipolar Permanent-Magnet Stepper Using  
4-Phase Excitation.

For counterclockwise rotation, power would be applied to the stator windings in reverse order as shown in studying Figure 3-27.



**Figure 3-27**  
Counterclockwise Rotation of A Bipolar Permanent-Magnet Stepper Using 4-Phase Excitation.

## VARIABLE RELUCTANCE STEPPER MOTOR

The primary difference between a variable reluctance stepper motor and a permanent magnet type is that the variable reluctance motor contains no magnet in the rotor. Instead, the rotor of the variable reluctance motor is constructed of soft iron and has salient poles. Like the permanent magnet motor, the number of teeth on the rotor and stator, as well as the number of winding phases, determines the step angle. Variable reluctance steppers are usually medium-step-angle devices (5 to 15 degrees) which can operate at high step speeds. Because the rotor is not magnetized, rotor position is independent of the polarity of stator pulse excitation; thus, a single-ended power source can be used. However, since the rotor contains no magnet there is no residual torque to maintain rotor position when the motor is deenergized.

Figure 3-28 shows a basic, variable reluctance stepper motor that has three stator windings A, B, and C; a soft iron rotor with four salient poles 1, 2, 3, and 4; and uses 3-phase excitation. In this example, if stator winding A is energized, rotor teeth 1 and 3 will align themselves with the axis of stator pole A, as seen in Figure 3-28A. For clockwise rotation, stator winding B would be energized and stator winding A deenergized, causing rotor poles 2 and 4 to move clockwise  $30^\circ$ ; thus, aligning themselves with the axis of stator pole B, as seen in Figure 3-28B. Granted, pole B will exert a magnetic attraction to both rotor poles 2 and 3, but the attraction for rotor pole 2 will be much greater since it is only  $30^\circ$  away, than for pole 3, which is  $60^\circ$  away; thus, the direction of rotation is assured.

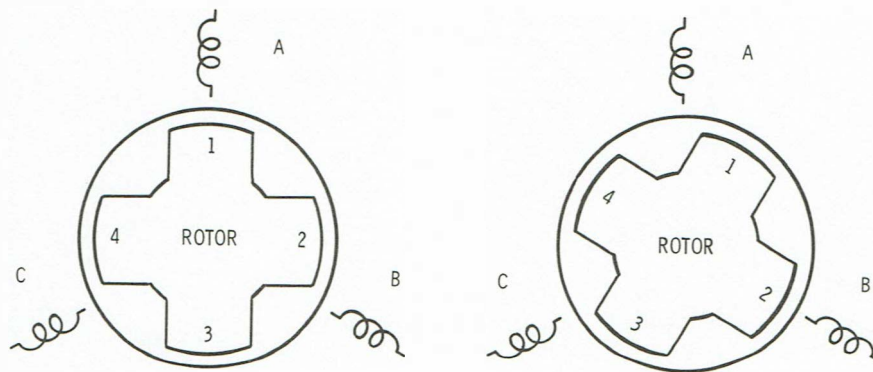


Figure 3-28  
Variable Reluctance Stepper Motor Using  
3-Phase Excitation.

If stator winding C is now energized and stator winding B deenergized, the rotor will move another  $30^\circ$  clockwise. Hence, if we continue to provide current pulses to the stator windings in the sequence A, B, C, A, etc., the motor will step  $30^\circ$  with each current pulse, requiring 12 such pulses to complete one revolution. Conversely, if the current pulses were supplied to the stator windings in the sequence of A, C, B, A, etc., the rotor would turn counterclockwise. To assure no rotor position loss or gain, one stator winding must be energized before excitation is removed from the other.



## BIFILAR STEPPING MOTOR

A bifilar stepping motor, commonly referred to as a unipolar stepping motor, is simply a variation of the permanent magnet stepping motor (sometimes called bipolar) previously discussed. However, in the bifilar (unipolar) stepper, each stator winding has been center-tapped, which in reality, places two stator windings on each stator pole. Since the stator windings are center-tapped, current can be sent through alternate halves to obtain alternate magnetic polarities.

The connections and switching sequence of a 2-phase bifilar (unipolar) stepping motor is shown in Figure 3-29. Because of this arrangement, a single 2-lead power supply can be used, thus, reducing the complexity of the control circuitry. However, for a bifilar (unipolar) motor to have the same number of turns per stator winding as a bipolar motor, the wire diameter must be decreased, and therefore, the resistance must be increased. As a result, a bifilar (unipolar) stepper has about 30% less torque than a bipolar stepper, at low step rates; but at higher step rates the torque outputs are approximately the same.

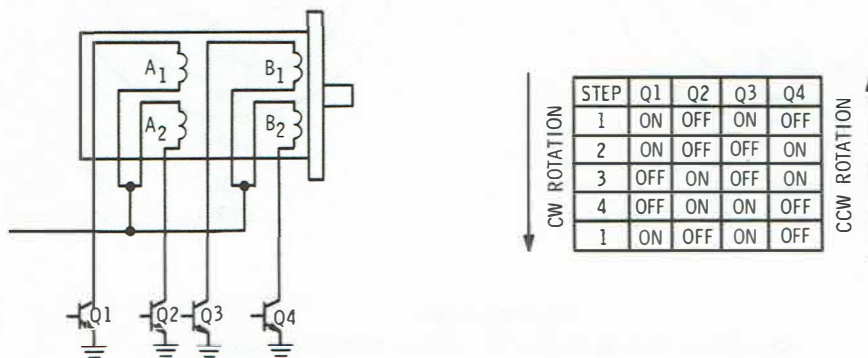
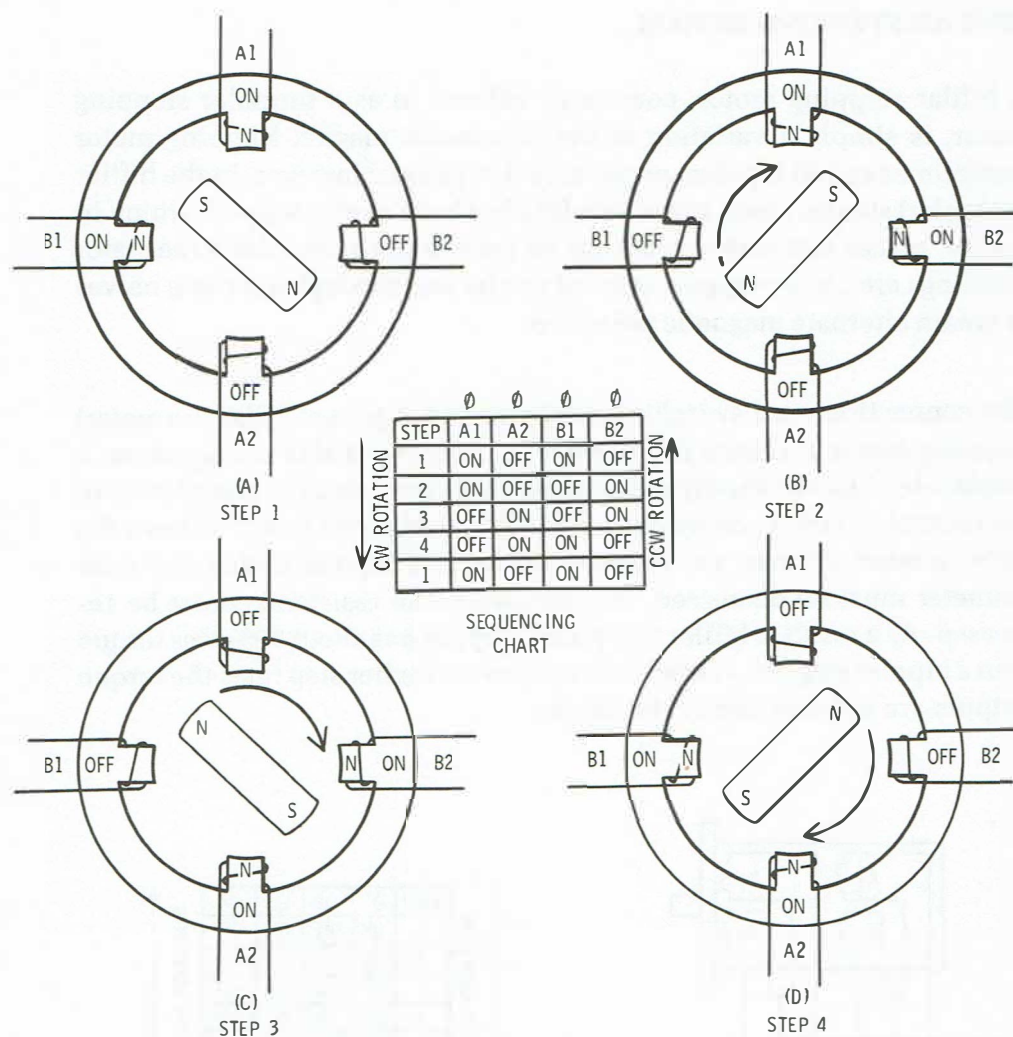


Figure 3-29  
Bifilar (Unipolar) 2-Phase Stepping Motor.



**Figure 3-30**  
Operation of A Basic Bifilar 2-Phase Stepping Motor.

The operation of a basic bifilar stepper motor is shown in Figure 3-30. For ease of understanding, the two phases are shown as four stator poles, and the rotor is shown with only a north and south pole. Actually, the bifilar stepper would have many more stator and rotor poles than shown in the Figure. In addition, in an actual bifilar motor, one set of phase windings would be placed behind the other, as shown in Figure 3-31.

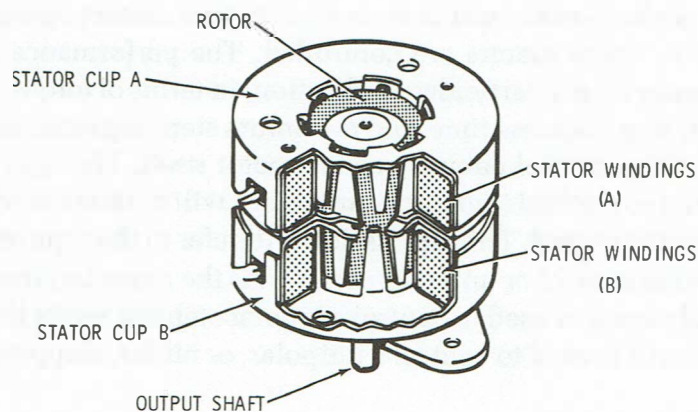


Figure 3-31  
2-Phase Bifilar Stepping Motor.

Refer to step 1 of the sequencing chart in Figure 3-30. If stator pole windings A1 and B1 became north stator poles when energized, the rotor would assume the position shown in Figure 3-30A. In step 2 on the sequencing chart, stator windings A1 and B2 are on and stator winding B1 is off; this moves the rotor 90° clockwise to the position shown in Figure 3-30B. To position the rotor another 90° clockwise, as shown in Figure 3-30C, stator winding A1 is deenergized, stator winding A2 is energized, and stator winding B2 remains energized; this action corresponds to step 3 in the sequencing chart. For an additional 90° of rotor rotation, refer to the sequencing chart and observe Figure 3-30D. To complete one revolution of the rotor, the stator windings would be energized in the same manner as they were for step 1.

## Stepper Motor Control

Now that we have examined how various stepper motors operate, we will discuss how these motors are controlled. The performance of a given stepper motor in any particular application; in terms of torque, speed, acceleration, step response time, and maximum stepping rates; is as much a function of the control circuitry as the motor itself. The type of stepper motor being controlled (bipolar or unipolar), will determine what type of control circuit is used. This discussion will refer to the type of control as either bipolar control or unipolar control. As the name implies, the bipolar control circuit is used to control a bipolar stepper; while the unipolar control circuit is used to control a unipolar, or bifilar, stepping motor.

### BIPOLAR CONTROL

The bipolar control circuit shown in Figure 3-32 is being used to drive a 2-phase bipolar permanent magnet stepping motor. As you recall, in a bipolar stepping motor, each motor phase consists of only one stator winding per pole. Therefore, both ends of the winding must be alternately connected to the voltage source to produce the correct magnetic stator field. Hence, four transistors are required per phase, or a total of eight transistors for a 2-phase motor.

Referring to Figure 3-32, if transistor switches Q1, Q4, Q5, and Q8 were turned on, current flow would be from the negative side of the battery through transistor Q4, **up** through winding A (creating a north pole), through transistor Q1, to the positive side of the battery. In addition, current would also flow from the negative side of the battery, through transistor Q8, **up** through winding B (creating a north pole), through transistor Q5, to the positive side of the battery. This would cause the axially-oriented rotor to rotate clockwise because of the action of the magnetic fields. This corresponds to step 1 in the sequencing chart.

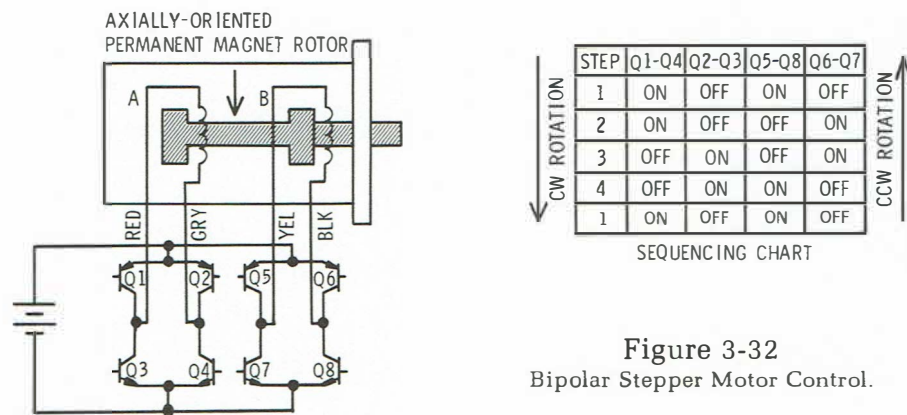


Figure 3-32  
Bipolar Stepper Motor Control.

In step 2, stator winding A remains a north pole (transistor switches Q1 and Q4 are still on), but now, transistor switches Q5 and Q8 have been turned off and transistor switches Q6 and Q7 have been turned on. This switching action causes current to flow from the negative side of the battery, through transistor Q7, **down** through winding B (now creating a south pole in winding B), through transistor switch Q6, to the positive side of the battery. This causes a magnetic field that will step the rotor clockwise once again.

To obtain an additional clockwise step, transistors Q6 and Q7 remain on (maintaining a south pole in winding B), transistors Q1 and Q4 are now turned off, and transistors Q2 and Q3 are turned on. This causes current to flow from the negative side of the battery, through transistor Q3, **down** through winding A (now creating a south pole in winding A), through transistor Q2, to the positive side of the battery. This corresponds to step 3 on the sequencing chart.

To complete the stepping action, the conditions of step 4 on the sequencing chart must be met. In this case, transistors Q2 and Q3 remain on (maintaining a south pole through winding A), transistors Q6 and Q7 are turned off, and transistors Q5 and Q8 are turned on. Current now flows from the negative side of the battery, through transistor Q8, **up** through winding B (again creating a north pole), through transistor Q5, to the positive side of the battery. To continue the clockwise rotation, the stepping must continue in the sequence 1, 2, 3, 4, etc.



### UNIPOLAR CONTROL

The unipolar control circuit shown in Figure 3-33 is used to drive a 2-phase bifilar (unipolar) permanent magnet stepping motor. Recalling, that in the bifilar (unipolar) stepper, each phase consists of two separate windings. The end of the first winding tied to the start of the second winding represents the center-tap. A bifilar winding has two coils wound on the same stator pole; therefore, the magnetic flux field is reversed by energizing one winding or the other from a single power source. The use of a bifilar stepper and unipolar control allows the drive circuit to be simplified. Not only are half as many power switches (transistors) required (4 vs 8), but the timing is not as critical to prevent a current short through two transistors, as is possible with a bipolar drive.

For clarity, in Figure 3-33, each half of the bifilar-wound stator coil has been labeled A1 and A2, and, B1 and B2. Assuming that the conditions of step 1 on the sequencing chart have been met, both transistor switches Q1 and Q3 are turned on. This allows current to flow from the negative side of the battery through switch Q1, into terminal 1 of the motor, **down** through stator winding A1 (creating a south pole in stator A), out of terminal 2 of the motor to the positive side of the battery. At the same time, current flows from the negative side of the battery through switch Q3, into terminal 5 of the motor, **down** through stator winding B1 (creating a south pole in stator B), out of terminal 6 of the motor to the positive side of the battery. This action creates a magnetic field that rotates the rotor clockwise one step.

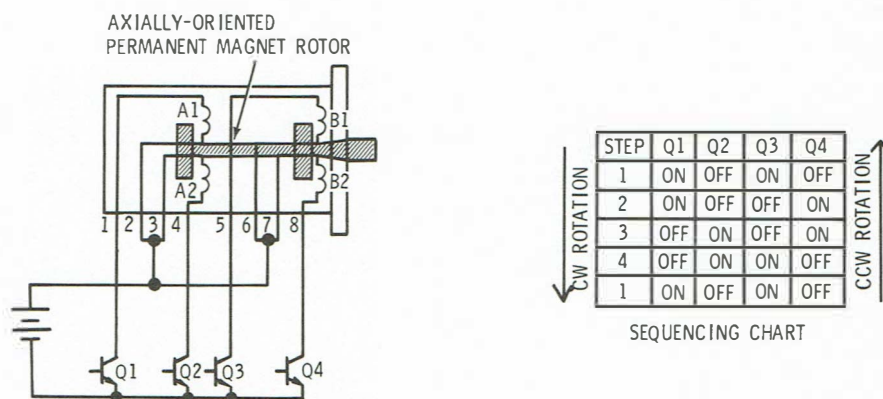


Figure 3-33  
Unipolar Stepper Motor Control.

Observing step 2 on the sequence chart, we see that switch Q1 is still on (creating a south pole through stator pole A), switch Q3 is off, and switch Q4 is now on. Now, current is allowed to flow **up** through stator winding B2; thus, creating a north pole at stator pole B. Due to the movement of the magnetic field, the rotor moves another step clockwise. By studying the sequencing chart and the schematic diagram, it becomes readily apparent that steps 3 and 4 will rotate the rotor further clockwise.

In either the bipolar or unipolar mode of control, you can make the rotor turn counterclockwise by simply reversing the stepping procedure shown on the sequencing chart. In other words, stepping sequence 1,2,3,4,1, etc. turns the rotor clockwise, so stepping sequence 1,4,3,2,1, etc. would turn it counterclockwise.

## Programmed Review

38. In stepper motors, feedback signals are not required as long as \_\_\_\_\_ integrity is maintained.

39. (pulse-to-step) In a stepper motor, rotor error \_\_\_\_\_ determined by the distance of angular travel. (is/is not)

40. (is not) In a permanent magnet stepping motor, the rotor and the \_\_\_\_\_ have a different number of teeth.

41. (stator) Step angle will be \_\_\_\_\_ by increasing (increased/decreased) the number of rotor and stator teeth.

42. (decreased) The variable reluctance stepping motor contains no \_\_\_\_\_ in the rotor.

43. (magnet) Compared to the permanent magnet stepping motor, the variable reluctance stepping motor has \_\_\_\_\_ holding torque. (more/less)

44. (less) The bifilar stepping motor has \_\_\_\_\_ windings on each stator pole.

45. (two) The \_\_\_\_\_ method of stepper motor control  
(bipolar/unipolar)  
is the most complex.

46. (bipolar) In a bipolar stepping motor, both ends of the winding must be \_\_\_\_\_ connected to the voltage source to provide the correct magnetic stator field.

47. (alternately) The unipolar control circuit has \_\_\_\_\_ as many switches as the bipolar control circuit.  
(twice/half)

48. (half) To change the direction of rotation of a motor from clockwise to counterclockwise in either the bipolar or unipolar control circuit, the stepping sequence must be \_\_\_\_\_.

(reversed)

## EXPERIMENT

Perform Experiment 4. You will find this experiment in Unit 12. After you finish the experiment, return to this unit and complete the Unit Examination.



## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct ones that appear after the exam.

1. One primary difference between a secondary cell and a primary cell is the fact that:
  - A. Primary cells produce a greater output.
  - B. Secondary cells may be recharged.
  - C. Primary cells may be recharged.
  - D. Secondary cells produce a greater output.
2. Which of the following is **not** a characteristic of a nickel-cadmium cell?
  - A. They can be charged at high rates.
  - B. They maintain an almost constant discharge voltage.
  - C. They can be used in any physical position.
  - D. They can only be stored for short periods of time.
3. The state of charge of a nickel-cadmium battery can be determined by:
  - A. Using a hydrometer.
  - B. Measuring the voltage across the terminals.
  - C. Measuring the current across the terminals.
  - D. None of the above.
4. The conservative estimate of the amount of capacity that can be drawn from a fully-charged cell or battery when it is discharged at a specific rate at a known temperature, to a specific cut off voltage is known as:
  - A. Rated capacity.
  - B. Specific capacity.
  - C. Discharge capacity.
  - D. Charge capacity.
5. Which charge rate is considered ideal for prolonging the life of a nickel-cadmium cell?
  - A. C/20 rate.
  - B. C/10 rate.
  - C. C/15 rate.
  - D. C/5 rate.

6. The most common method of monitoring a nickel-cadmium cell during rapid charge is:
  - A. Temperature monitoring.
  - B. Pressure monitoring.
  - C. Voltage monitoring.
  - D. Current monitoring.
7. Which of the following is **not** a characteristic of a gelled-electrolyte battery?
  - A. The ability to retain a charge during prolonged storage.
  - B. The ability to recover from deep discharges.
  - C. An open-circuit voltage of 1.2 volts per cell.
  - D. Lack of "memory".
8. Which of the following gelled-electrolyte battery ratings would produce the greatest discharge rate?
  - A. C/20.
  - B. C/5.
  - C. 3C
  - D. C/1.
9. To charge a gelled-electrolyte battery, the charging voltage must be high enough to:
  - A. Overcome the heat build-up inside the battery.
  - B. Overcome the high internal resistance of the battery.
  - C. Compensate for internal pressure build-up.
  - D. Compensate for small variations between cells.
10. In a gelled-electrolyte battery, which of the following is not considered to be a permanent failure?
  - A. Internal short within a cell.
  - B. Open circuit within the battery.
  - C. Drying out of the sealed electrolyte.
  - D. High temperature operation.
11. Which of the following types of motors has the greatest starting torque?
  - A. DC series-wound motor.
  - B. 3-phase AC motor.
  - C. DC shunt-wound motor.
  - D. Split-phase AC motor.

12. The main function of the commutator in a field-wound DC motor is to:
- A. Provide a constant polarity current to the armature windings.
  - B. Reverse the current through the armature windings.
  - C. Produce a rotating magnetic field in the stator.
  - D. Hold the motor brushes in place.
13. Which type of motor can most easily be reversed by switching the applied voltage from one field coil to the other?
- A. Series-wound motor.
  - B. Compound-wound motor.
  - C. Split-series-wound motor.
  - D. Shunt-wound motor.
14. Which type of speed control is only suitable for obtaining speeds greater than the base speed?
- A. Armature-voltage control.
  - B. Stator-rotor current control.
  - C. Shunt-field control.
  - D. Series-field control.
15. Which of the following is **not** a characteristic of a DC brushless motor?
- A. A commutator is not required.
  - B. It can be operated in a phase-locked loop.
  - C. Once operating speed has been reached, it can be switched to synchronous operation.
  - D. It develops a large starting torque.
16. In a Hall-effect generator, which of the following semiconductor material factors cannot be changed?
- A. Mechanical pressure.
  - B. Temperature.
  - C. Magnetic field intensity.
  - D. Amount of current flow.
17. In a DC stepper motor, which of the following factors cannot be controlled?
- A. Speed of rotation.
  - B. Direction of rotation.
  - C. Number of incremented steps.
  - D. They can all be controlled.

18. Which of the following stepper motors would have the least amount of error?
- A. A 240-steps-per-revolution motor.
  - B. A 144-steps-per-revolution motor.
  - C. A 72-steps-per-revolution motor.
  - D. A 24-steps-per-revolution motor.
19. Which of the following stepper motors provides the **least** amount of residual torque?
- A. Bipolar permanent-magnet stepping motor.
  - B. Variable reluctance stepping motor.
  - C. Unipolar permanent-magnet stepping motor.
  - D. They all provide the same amount of residual torque.
20. Which of the following types of stepper motor and control circuits would be the easiest to control?
- A. Variable reluctance motor using bipolar control.
  - B. Bipolar permanent-magnet motor using bipolar control.
  - C. Variable reluctance motor using bifilar control.
  - D. Unipolar permanent magnet motor using unipolar control.

## EXAMINATION ANSWERS

For your convenience, the page where you can find the correct answer is shown following the answer.

1. B — Secondary cells may be recharged. [3-9]
2. D — They can only be stored for short periods of time. [3-11]
3. D — None of the above. [3-13]
4. A — Rated Capacity. [3-19]
5. C — C/15 rate. [3-22]
6. A — Temperature monitoring [3-24]
7. C — An open-circuit voltage of 1.2 volts per cell. [3-31]
8. B — 3C rate. [3-36]
9. D — Compensate for small variations between cells. [3-41]
10. D — High temperature operation. [3-48]
11. A — DC series-wound motor. [3-53,57]
12. B — Reverse the current through the armature windings. [3-53]
13. C — Split-series motor. [3-57]
14. C — Shunt field control. [3-58]
15. D — It develops a large starting torque. [3-63]
16. A — Mechanical pressure. [3-65]
17. D — They can all be controlled. [3-69]
18. A — A 240-steps-per-revolution motor. [3-69]
19. B — A variable reluctance stepping motor. [3-75]
20. D — Unipolar permanent-magnet motor using unipolar control. [3-82]





*Unit 4*

**MICROPROCESSOR  
FUNDAMENTALS**

## CONTENTS

Introduction .....	4-3
Unit Objectives .....	4-5
Unit Activity Guide .....	4-6
Terms and Conventions .....	4-7
An Elementary Microcomputer .....	4-14
Executing a Program .....	4-27
Addressing Modes .....	4-42
Experiment 5 .....	4-65
Binary Arithmetic .....	4-66
Two's Complement Arithmetic .....	4-86
Boolean Operations .....	4-97
Experiment 6 .....	4-106
Unit Examination .....	4-107
Examination Answers .....	4-113

## INTRODUCTION

As stated in Unit 1, all of today's more sophisticated robots use some sort of computer control — the ET-18 Robot Trainer is no exception. Because it is very important that you thoroughly understand this type of control, three units have been devoted to the subjects of microcomputers and microprocessors.

In this Unit, you will be introduced to basic microprocessor principles. In addition, you will learn how a microprocessor performs basic number operations using a special type of "Computer Arithmetic."

A microprocessor is a very complex electronic circuit. It consists of thousands of microscopic transistors which are integrated on a tiny chip of silicon that is often no more than one-eighth inch square. The chip is usually placed in a package containing 40 or more leads.

The thousands of transistors that make up the microprocessor are arranged to form many different circuits within the chip. From the standpoint of learning how the microprocessor operates, the most important circuits on the chip are registers, counters, and decoders. You will learn how these circuits work together to perform simple but useful tasks.

Like any other technology, the field of microprocessors has its own terminology. Therefore, a great deal of information contained in this Unit pertains to the definition of these terms.

Since microprocessors use binary numbers for data and control, it is important that you become familiar with them. Computer arithmetic involves many forms of number manipulation. You will be shown the fundamentals of binary mathematics: addition, subtraction, multiplication, and division. Then you will learn to perform two's complement arithmetic using binary numbers. Finally, you will be shown how the microprocessor performs the four basic Boolean logic operations.

In order to get the most out of this Unit, you need a working knowledge of number systems. If you are not familiar with binary, decimal, octal, and hexadecimal notation, or perhaps just need a refresher, refer to Appendix A at the end of the text. This Appendix contains a short discussion of number systems.

Use the following “Unit Objectives” to evaluate your progress. When you can accomplish all of them, you will have successfully completed this Unit. You can use the “Unit Activity Guide” to keep a record of those sections that you have completed.



## UNIT OBJECTIVES

When you complete this Unit, you will be able to:

1. Define the following terms: microprocessor, microcomputer, input, output, I/O, I/O devices, I/O port, instruction, program, stored program, word, byte, MPU, ALU, operand, memory, address, read, write, RAM, fetch, execute, MPU cycle, mnemonic, opcode, and bus.
2. Explain the purpose of the following microprocessor circuits: accumulator, program counter, instruction decoder, controller-sequencer, data register, and address register.
3. Describe the difference between inherent, immediate, and direct addressing.
4. Explain the data flow that takes place between the various circuits of a hypothetical microprocessor during the execution of a simple program.
5. Perform addition, subtraction, multiplication, and division functions, using binary numbers.
6. Derive the one's complement of a binary number.
7. Derive the two's complement of a binary number.
8. Manipulate binary numbers using AND, OR, and Exclusive OR operations.
9. Logically invert binary numbers.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the section on "Terms and Conventions."	_____
<input type="checkbox"/> Answer Programmed Review Questions 1 — 9.	_____
<input type="checkbox"/> Read the section on "An Elementary Microprocessor."	_____
<input type="checkbox"/> Answer Programmed Review Questions 10 — 20.	_____
<input type="checkbox"/> Read the section on "Executing a Program."	_____
<input type="checkbox"/> Answer Programmed Review Questions 21 — 26.	_____
<input type="checkbox"/> Read the section on "Addressing Modes."	_____
<input type="checkbox"/> Answer Programmed Review Questions 27 — 32.	_____
<input type="checkbox"/> Perform Experiment 5	_____
<input type="checkbox"/> Read the section on "Binary Arithmetic."	_____
<input type="checkbox"/> Answer Programmed Review Questions 33 — 43.	_____
<input type="checkbox"/> Read the section on "Two's Complement Arithmetic."	_____
<input type="checkbox"/> Answer Programmed Review Questions 44 — 53.	_____
<input type="checkbox"/> Read the section on "Boolean Operations."	_____
<input type="checkbox"/> Answer Programmed Review Questions 54 — 62.	_____
<input type="checkbox"/> Perform Experiment 6	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## TERMS AND CONVENTIONS

Before we proceed with our discussion of microprocessors and mini-computers, let's take a few moments and study the terms associated with these devices.

A **microprocessor** is a logic device that is used in many digital electronic systems. It is also being used by hobbyists, experimenters and as a general-purpose computer for robot control. However, there is a difference between the microprocessor and the microcomputer.

The microprocessor unit, or MPU, is a complex logic element that performs arithmetic, logic, and control functions. The trend is to package it as a single integrated circuit. A **microcomputer** contains a microprocessor, but it also contains other circuits such as memory devices to store information, interface adapters to connect it with the outside world, and a clock to act as a master timer for the system. Microcomputers are used to control one or more robots.

Figure 4-1 shows a typical microcomputer in which these additional circuits are added. The arrows between the clock and the MPU represent conductors over which binary information flows. The wide arrows represent conductors over which binary information flows. The wide arrows represent several conductors connected in parallel. A group of parallel conductors that carry information is called a **bus**.

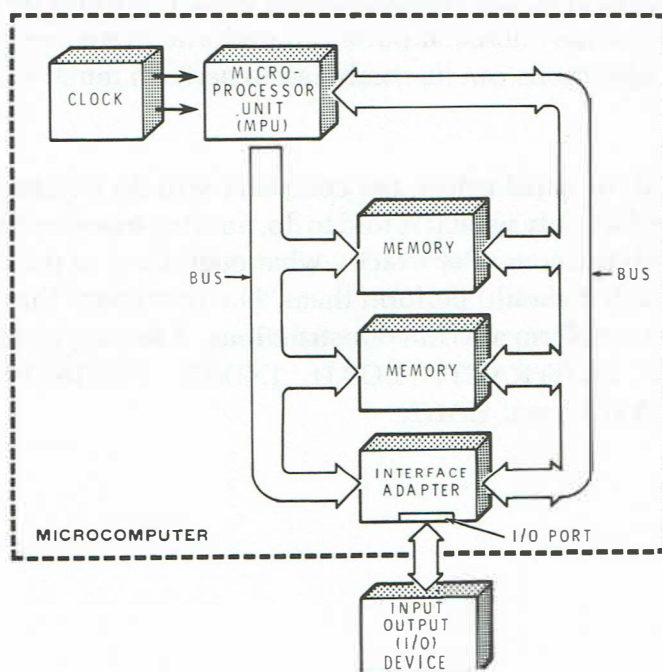


Figure 4-1  
A basic microcomputer.

In Figure 4-1, the microcomputer is comprised of everything inside the dotted line; while everything outside the dotted line is referred to as the **outside world**. All microcomputers must have some way of communicating with the outside world. Information received by the microcomputer from the outside world is referred to as **input** data; whereas information transmitted to the outside world from the microcomputer is referred to as **output** data.

Input information may come from devices like paper tape readers, typewriters, mechanical switches, keyboards, or even other computers. Output information may be sent to video displays, output typewriters, paper-tape punchers, or line printers. Some devices such as the teletypewriter can serve as both an input and an output device. These devices are referred to as **input/output** or **I/O** devices. The point at which the I/O device connects to the microcomputer is called an **I/O port**. In the case of microcomputer controlled robots, much of the input information comes from sensing devices, and much of the output information is used to control various robotic functions.

## Stored Program Concept

A microcomputer is capable of performing many different operations. It can add and subtract numbers, perform logic operations, read information from an input device, and transmit information to an output device. In fact, depending on the microprocessor used, there may be 100 or more operations that the microcomputer can perform. Moreover, two or more individual operations can be combined to perform much more complex operations.

In spite of all its capabilities, the computer will do nothing on its own accord. It will do only what it is told to do, nothing more and nothing less. You must tell the computer exactly what operations to perform and the order in which it should perform them. The operations that a computer can be told to perform are called **instructions**. A few typical instructions are **ADD**, **SUBTRACT**, **LOAD INDEX REGISTER**, **STORE ACCUMULATOR**, and **HALT**.

A group of instructions that allow the computer to perform a specific job is called a **program**. One who writes these instructions is called a **programmer**. To design with microprocessors, the engineer must become a programmer. To repair microprocessor-based equipment, the technician must understand programming. Generally, the length of the program is proportional to the complexity of the task the computer is to perform. A program for adding a list of numbers may require only a dozen instructions. On the other hand, a program for controlling a robot during welding operations may require hundreds of instructions.

A computer is often compared to a calculator that is told what to do by the operator via the keyboard. Even inexpensive calculators can perform several operations that can be compared to instructions in the computer. By pressing the correct keys, in the correct sequence, you can instruct the calculator to add, subtract, multiply, divide, and clear the display. Of course, you must also enter the numbers that are to be added, subtracted, etc. With the calculator, you can add a list of numbers as quickly as you can enter the numbers and the instructions. Therefore, the operation is limited by the speed and accuracy of the operator.

From the start, computer designers recognized that it was the human operator that slowed the computation process. To overcome this, they developed the **stored program concept**. Using this approach, the program is stored in the computer's memory. Suppose, for example, that you have 20 numbers that are to be manipulated by a program that is composed of 100 instructions. Let's further suppose that 10 answers will be produced in the process.

Before any computation begins, the 100-instruction program plus the 20 numbers are loaded into the computer's memory. Furthermore, 10 memory locations are reserved for the answers. Only then is the computer allowed to execute the program. The actual computation time might be less than one millisecond. Compare this to the time it would take to manually enter the instructions and numbers, one at a time, while the computer is running. This automatic operation is one of the features that distinguishes the computer from the simple, inexpensive calculator.



## Computer Words

In computer terminology, a **word** is a group of binary digits that can occupy a storage location. Although the word may be made up of several binary digits, the computer handles each word as if it were a single unit. Thus, the word is the fundamental unit of information used in the computer.

A word may be a binary number that is to be handled as data. Or, the word may be an instruction that tells the computer which operation it is to perform. It may be an ASCII character representing a letter of the alphabet (Appendix A has a detailed description of ASCII characters). Finally, a word can be an “address” that tells the computer where a piece of data is located.

## Word Length

In the past few years, a wide variety of microprocessors have been developed. Their cost, characteristics, and capabilities vary widely. One of the most important characteristics of any microprocessor is the word length it can handle. This refers to the length (in bits) of the most fundamental unit of information.

The most common word length for microprocessors is 8 bits. In these units; numbers, addresses, instructions, and data are represented by 8-bit binary numbers. The lowest 8-bit binary number is 0000 0000<sub>2</sub> or 00<sub>16</sub> (Hexadecimal). The highest is 1111 1111<sub>2</sub> or FF<sub>16</sub>. Translated to decimal, this range is from 0 to 255. Thus, an 8-bit binary number can have any one of 256 unique values.



An 8-bit word can specify positive numbers between 0 and 255 or, if the 8-bit word is an instruction, it can specify any of 256 possible operations. It is also entirely possible that the 8-bit word is an ASCII character. In this case, it can represent letters of the alphabet, punctuation marks, or numerals. As you can see, the 8-bit word can represent many different things, depending on how it is interpreted. The programmer must insure that an ASCII character or binary number is not interpreted as an instruction.

While the 8-bit word length is still quite popular, other word lengths are sometimes used. The earliest microprocessors used a 4-bit word length, and 4-bit microprocessors are still used in many cases because of their low cost. They are ideally suited for control operations that require few command signals. However, the 16-bit microprocessor has become increasingly popular.

Longer word lengths allow us to work with larger numbers. For example, a 16-bit word can represent decimal numbers up to 65,535. But, this capability adds to the complexity and cost of the microprocessor. Since most microprocessors use 8-bit word lengths, we will restrict our discussion to devices of this type, but even though the word length is 8-bits, it does not mean that we are restricted to decimal numbers below 256. It simply means that you must use two or more words to represent larger numbers.

The 8-bit word length defines the size of many different components in the microprocessor system. For example, many of the important registers will have 8-bit capacity. Memory will be capable of holding a large number of 8-bit words, and the bus, which is used to transfer data words, will consist of eight parallel conductors.

Even 16-bit microprocessors use 8-bit segments of data in many applications. For example, inputs from teletypewriters often consist of 8-bit ASCII characters. To distinguish these 8-bit segments of information from the 16-bit (or longer) word lengths, another term has come into general use: the term **byte**. A byte is a group of bits that are handled as a single unit. Generally a byte is understood to consist of 8-bits. In the 8-bit microprocessor, each word consists of one byte. However, in the 16-bit machines, each word contains two bytes. Figure 4-2 illustrates these points.

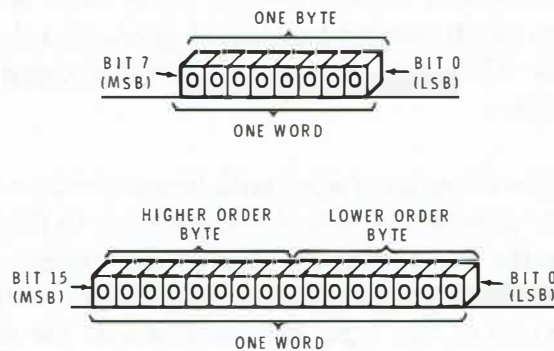


Figure 4-2  
Words and bytes.

Figure 4-2 also shows how the bits that make up the computer word are numbered. The least significant bit (LSB) is on the right while the most significant bit (MSB) is on the left. In the 8-bit word, the bits are numbered 0 through 7 from right to left; while in the 16-bit word, the bits are numbered 0 through 15 as shown. The lower 8 bits are called the lower order byte, while the upper 8 bits are called the higher order byte.

## Programmed Review

1.	In a microcomputer, a group of parallel conductors that carry information is called a _____.
2.	(bus) Information received by a microcomputer from the outside world is referred to as _____ data.
3.	(input) A device such as a _____ (teletypewriter/video display) can serve as both an input and an output device.
4.	(teletypewriter) A _____ is a group of instructions that allow the computer to perform a specific task.
5.	(program) A "stored program" is stored in the computer's _____.
6.	(memory) The _____ is the fundamental unit of information used in the computer.
7.	(word) An 8-bit binary number can represent decimal numbers from 0 through _____.
8.	(255) A _____ is a group of bits, generally 8, that are handled as a single unit.
9.	(byte) In an 8-bit binary number, the _____ (LSB/MSB) is on the right and carries the _____ (least/most) weight.
(LSB/least)	

## AN ELEMENTARY MICROCOMPUTER

As you study microcomputers, microprocessors (their main component) may be the most difficult part to understand, since they are very complex devices. A microprocessor may have a dozen or more registers (temporary storage locations) varying in size from 1 bit to 16 bits. It will have scores of instructions, most of which can be implemented in several different ways. It will have data, address, and control busses. In short, it can be very intimidating to start out by studying an advanced microprocessor. It is better to start with a “stripped down” version. Without the advanced features, the device is easier to understand, yet it retains the characteristics of an actual microprocessor.

The microprocessor developed in this unit is hypothetical in nature, but it is so close to the real thing that the programs we develop for it will actually run on the ET-18 Robot Trainer. Also, as you will see later, one of the most popular microprocessors in use today is merely a vastly advanced version of our elementary model.

A block diagram of a basic microcomputer is shown in Figure 4-3, which shows the microprocessor, the memory, and the I/O circuitry. Ignore the I/O circuitry at this time; just assume that the program and data are already in memory, and that the results of any computations will be held in a register or stored in memory. Of course, the program and data actually must come from the outside world, and the results must be sent to the outside world. We will save these procedures until later so you can concentrate on the microprocessor unit and the memory.

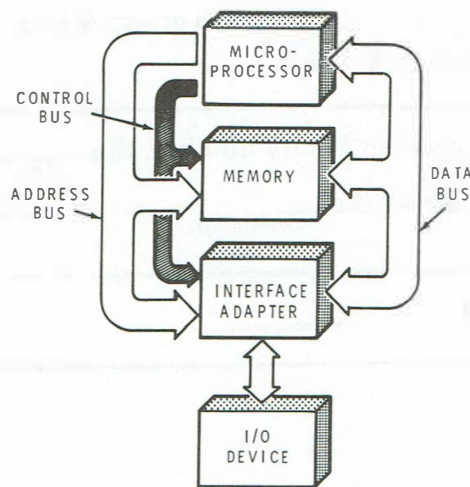


Figure 4-3  
The basic microcomputer.

## The Microprocessor Unit (MPU)

The microprocessor unit, referred to as the MPU, is shown in greater detail in Figure 4-4. For simplicity, only the major registers and circuits are shown. In our elementary unit, most of the counters, registers, and busses are 8 bits wide. That is, they can accommodate 8-bit, or 1-byte words.

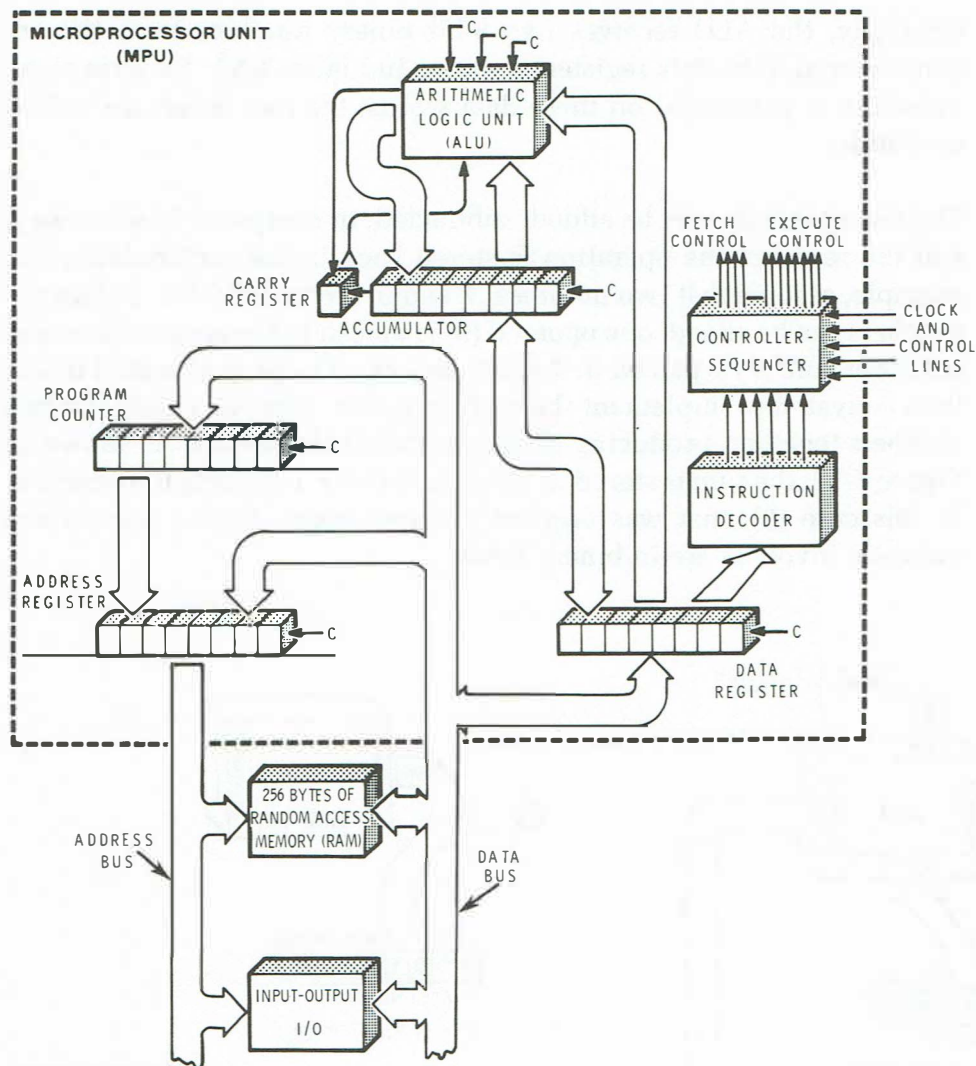


Figure 4-4  
An elementary microprocessor.



One of the most important circuits in the microprocessor is the **arithmetic logic unit (ALU)**. Its function is to perform arithmetic or logic operations on the data words that are delivered to it. The ALU has two main inputs. One comes from a register called the accumulator, and the other comes from the data register. The ALU can add the two input data words together, or it can subtract one from the other. It can also perform some logic operations which will be discussed later. The operation that the ALU performs is determined by the signals on the various control lines (marked C on the block diagram).

Generally, the ALU receives two 8-bit binary numbers from the accumulator and the data register as shown in Figure 4-5A. Because some operation is performed on these data words, the two inputs are called **operands**.

The two operands may be added, subtracted, or compared in some way, and the result of the operation is stored back in the accumulator. For example, assume that two numbers, 7 and 9, are to be added. Before the numbers can be added, one operand (9) is placed in the accumulator; the other operand (7) is placed in the data register. The proper control line is then activated to implement the add operation. The ALU adds the two numbers together, producing their sum (16) at the output. As shown in Figure 4-5B, the sum is stored in the accumulator, replacing the operand, in this case (9), that was originally stored there. Notice that all the numbers involved are in binary form.

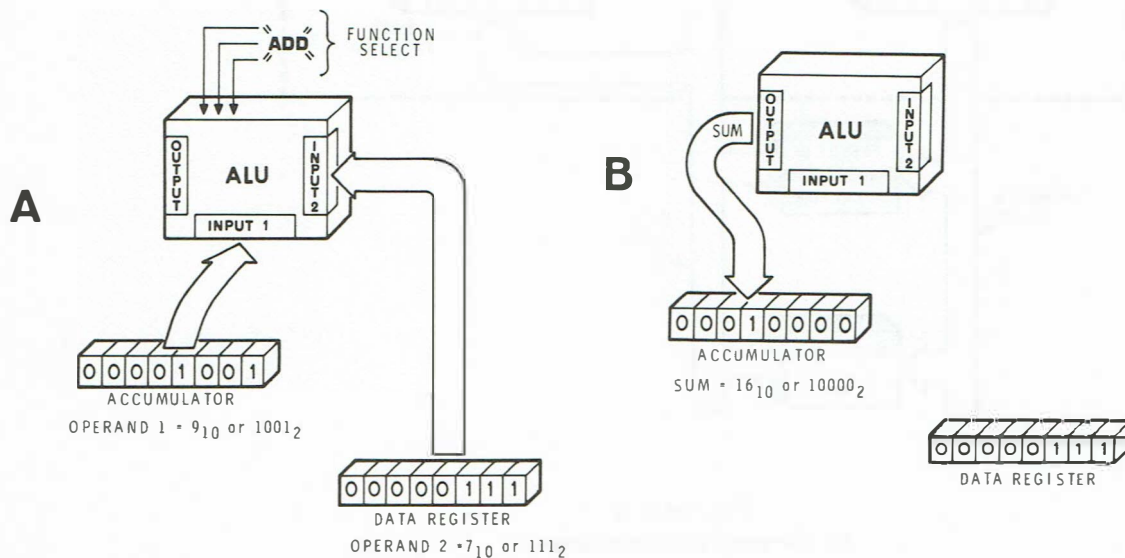


Figure 4-5  
An arithmetic logic unit (ALU).



The **accumulator** is the most useful register in the microprocessor. During arithmetic and logic operations, it performs a dual function. Before the operation, it holds one of the operands. After the operation, it holds the resulting sum, difference, or logical answer. The accumulator receives several instructions in every microprocessor. For example, the “load accumulator” instruction causes the contents of some specified memory location to be transferred to the accumulator. The “store accumulator” instruction causes the contents of the accumulator to be stored at some specified location in memory.

The **data register** is a temporary storage location for the data going to or coming from the data bus. For example, it holds an instruction while the instruction is being decoded. Also, it holds a data byte while the word is being stored in memory. The MPU also contains several other important registers and circuits: the address register, the program counter, the instruction decoder, and the controller-sequencer. (Refer back to Figure 4-4.)

The **address register** is another temporary storage location. It holds the address of the memory location or I/O device that is used in the operation presently being performed.

The **program counter** controls the sequence in which the instructions in a program are performed. Normally, it does this by counting the sequence 1, 2, 3, 4, etc. At any given instant, the count indicates the location in memory from which the next byte of information is to be taken.

The **instruction decoder** does just what its name implies. After an instruction is pulled from memory and placed in the data register, this circuit decodes the instruction. The decoder examines the 8-bit code and decides which operation is to be performed.

The **controller-sequencer** produces a variety of control signals to carry out the instruction. Since each instruction is different, a different combination of control signals is produced for each instruction. This circuit determines the sequence of events necessary to complete the operation described by the instruction.

Later you will see how these various circuits work together to execute simple programs. But first, take a closer look at the memory for our microcomputer.

## Memory

A simplified diagram of the 256-word, 8-bit read/write memory that is used in our hypothetical microcomputer is shown in Figure 4-6. The memory consists of  $FF_{16}$ , or  $256_{10}$  locations, each of which can store an 8-bit word. This size memory is often referred to as  $256 \times 8$ . A read/write memory is one in which data can be written in and read out with equal ease.

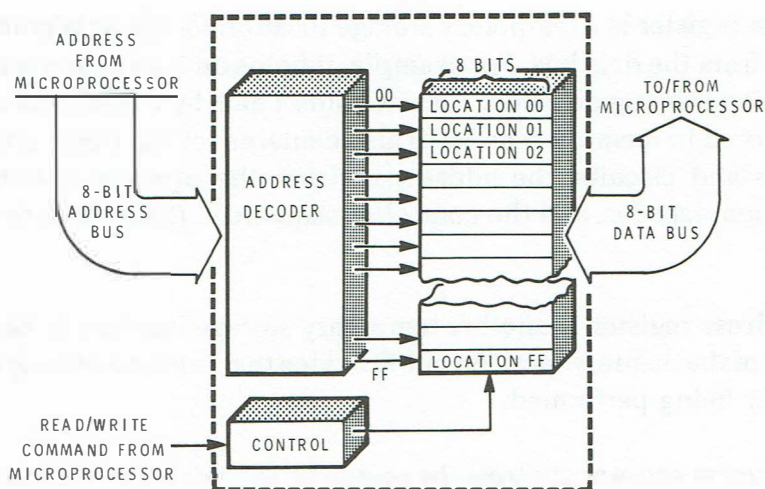


Figure 4-6  
The random access memory.

Two busses and a number of control lines connect the memory with the microprocessing unit. The address bus will carry an 8-bit binary number, which can specify up to  $256_{10}$  locations, from the MPU to the memory address decoder. Each location is assigned a unique number, called its address. The first location is given the address 0, and the last location is given the address  $255_{10}$ , which is 1111 1111 in binary and FF in hexadecimal. A specific location is selected by placing its 8-bit address on the address bus. The address decoder decodes the 8-bit number and selects the proper memory location.

The memory also receives a control signal from the MPU. This signal tells the memory which operation is to be performed. A **read** signal indicates that the selected location is to be read out. This means that the 8-bit number contained in the selected location is to be placed on the data bus where it can be transferred to the MPU.

This procedure is illustrated in Figure 4-7. Assume that the MPU is to read out the contents of memory location  $04_{16}$ . Let's further assume that the number stored there is  $97_{16}$ . First, the MPU places the address  $04_{16}$  on the address bus. The decoder recognizes the address and selects the proper memory location. Second, the MPU sends a READ signal to the memory, indicating that the contents of the selected location are to be placed on the data bus. Third, the memory responds by placing the number  $97_{16}$  on the data bus. The MPU can then pick up the number and use it as needed.

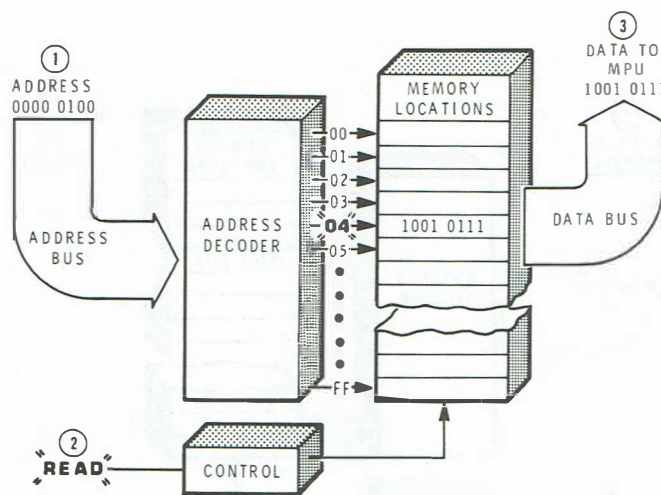


Figure 4-7  
Reading from memory.

It should be pointed out that the process of reading out of a memory location does not disturb the contents of that location. That is, the number  $97_{16}$  will still be present at memory location 04 after the read operation is finished. This characteristic is referred to as nondestructive readout (NDRO). It is an important feature because it allows us to read out the same data as many times as needed.

The MPU can also initiate a WRITE operation. This procedure is illustrated in Figure 4-8. During a WRITE operation, a data word is taken from the data bus and placed in the selected memory location. As an example, let's see how the MPU can store the number  $52_{16}$  at location 03. First, the MPU places the address 03 on the address bus. The decoder responds by selecting memory location 03. Second, the MPU places the number  $52_{16}$  on the data bus. Third, the MPU sends the WRITE signal. The memory responds by storing the number contained on the data bus in the selected location. That is,  $52_{16}$  is stored in location 03. The previous contents of the selected location are lost as the new number is written in that location. The accepted name for a memory of this type is **random access memory**, or **RAM**.

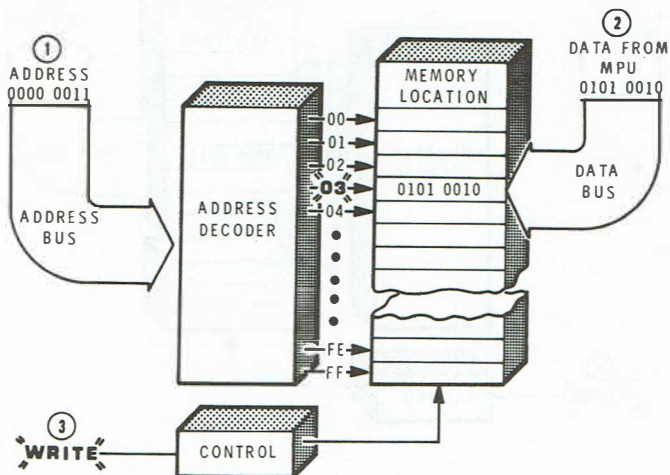


Figure 4-8  
Writing into memory.

However, there are actually two classifications of memory which must be considered in order to understand the microcomputer. The first type of memory, RAM, which we just discussed, is randomly accessible. That is, it does not have to be addressed in sequential order. But, due to convention, only **read/write** memory is called RAM. In any case, RAM acts as a temporary storage location within the microcomputer, for either programs or data. Any program you wish to run, or data that you wish to use, must first be loaded into RAM. Once in RAM, programs and data may be altered by the user.

A second type of memory, **read only memory** (ROM), is a permanent storage area for programs or constants that are essential for the operation of the microcomputer. As the name implies, information can only be read from this type of memory. The contents of a read only memory are protected from any accidental write operations.



Generally, you will not program a ROM; it is usually done at the time of manufacture, and each ROM is designed for a specific use with a specific system. It is handy to know, however, that other ROMs may be available that will work with a given microprocessor. Both RAM and ROM have their own particular uses, but as a programmer or operator, you will deal primarily with RAM.

## Fetch-Execute Sequence

When the microcomputer is executing a program, it goes through a fundamental sequence that is repeated over and over again. Recall that a program consists of instructions that tell the microcomputer exactly what operations to perform. These instructions must be stored in an orderly manner in memory. Instructions must be “fetched”, one at a time from memory, by the MPU. The MPU then “executes” the instruction.

The operation of the microcomputer can be broken down into two phases, as shown in Figure 4-9. When the microprocessor is originally started, it enters the FETCH PHASE. During the fetch phase, an instruction is taken from memory and decoded by the MPU. Once the instruction is decoded, the MPU switches to the EXECUTE PHASE. During this phase, the MPU carries out the operation dictated by the instruction.

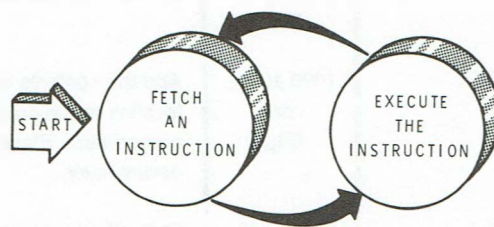


Figure 4-9  
The fetch-execute sequence.

The fetch phase always consists of the same series of operations. Thus, it always takes the same amount of time. However, the execute phase will consist of different sequences of events, depending on what type of instruction is being executed. Consequently, the time of the execute phase may vary considerably from one instruction to the next.



## A Sample Program

Now that you have a general idea of the registers and circuits in a microcomputer, we will examine how all these circuits work together to execute a simple program. At this point, we are primarily interested in how the microcomputer operates; therefore, the program will be a very simple one.

As an example, let's see how the computer goes about solving a problem like  $7 + 10 = ?$ . While this may seem like an incredibly easy problem, the computer, if left to its own resources, does not have the foggiest notion of how to solve it. You must tell the computer exactly how to solve the problem, right down to the smallest detail, by writing a program.

However, before you can write the program, you must know what instructions are available to you and the computer. Every microprocessor comes with a listing of its **instruction set**. Assume that, after looking over the list, you decide that three instructions are necessary to solve the problem. These three instructions and a description of what they do are shown in Figure 4-10.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	1000 0110 <sub>2</sub> or 86 <sub>16</sub>	Load the contents of the next memory location into the accumulator.
Add	ADD	1000 1011 <sub>2</sub> or 8B <sub>16</sub>	Add the contents of the next memory location to the present contents of the accumulator. Place the sum in the accumulator.
Halt	HLT	0011 1110 <sub>2</sub> or 3E <sub>16</sub>	Stop all operations.

Figure 4-10  
Instructions used in the sample program.

The first column in the table gives the name of the instruction. When writing programs, it is often inconvenient to write out the entire name. For this reason, each instruction is given an abbreviation or a memory aid called a **mnemonic** (pronounced ne-mŏn'ik). The mnemonic abbreviation for each instruction is given in the second column. The third column is called the operation code or **opcode**. This is the binary number that the computer and the programmer use to represent the instruction. The opcode is given in both binary and hexadecimal form. The final column describes exactly what operation is performed when the instruction is executed. Study this table carefully; you will be using these instructions over and over again.

Since we have decided to add 7 to 10 in our elementary program and place the sum in the accumulator, we will proceed as follows:

- First, load 7 into the accumulator with the LDA instruction.
- Next, add 10 to the accumulator, using the ADD instruction.
- Finally, stop the computer with the HLT instruction.

Using the mnemonics and the decimal representations of the numbers to be added, the program looks like this:

```
LDA 7  
ADD 10  
HLT
```

Unfortunately, the basic microprocessor cannot understand mnemonics or decimal numbers. It can interpret binary numbers and nothing else. Thus, you must write the program as a sequence of binary numbers. You can do this by replacing each mnemonic with its corresponding opcode and each decimal number with its binary counterpart.

That is:

LDA 7 becomes 1000 0110          0000 0111  
(opcode from Figure 4-10) (binary representation for 7)

And:

ADD 10 becomes 1000 1011          0000 1010  
(opcode from Figure 4-10) (binary representation for 10)

Finally,

HLT Becomes 0011 1110  
(opcode from Figure 4-10)

Notice that the program consists of three instructions. The first two instructions have two parts: an 8-bit opcode followed by an 8-bit operand. The operands are the two numbers that are to be added, in this case, 7 and 10.

Recall that the microprocessor and memory work with 8-bit words or bytes. Because the first two instructions consist of 16-bits of information, they must be broken into two 8-bit bytes before they can be stored in memory. Thus, when the program is stored in memory, it will look like this:

1st Instruction	{ 1000 0110 Opcode for LDA 0000 0111 Operand (7)
2nd Instruction	{ 1000 1011 Opcode for ADD 0000 1010 Operand (10)
3rd Instruction	0011 1110 Opcode for HLT

Five bytes of memory are required. You can store this 5-byte program any place you like. Assuming you store it at the first five memory locations, the memory can be diagrammed as shown in Figure 4-11.

ADDRESS		MEMORY	MNEMONIC/DECIMAL CONTENTS
HEX	BINARY	BINARY CONTENTS	
00	0000 0000	1 0 0 0 0 1 1 0	LDA
01	0000 0001	0 0 0 0 0 1 1 1	7
02	0000 0010	1 0 0 0 1 0 1 1	ADD
03	0000 0011	0 0 0 0 1 0 1 0	10 <sub>10</sub>
04	0000 0100	0 0 1 1 1 1 1 0	HLT
	.		
	.		
	.		
	.		
	.		
FD	1111 1101		
FE	1111 1110		
FF	1111 1111		

Figure 4-11  
The program in memory.

Notice that each memory location has two 8-bit binary numbers associated with it. One is its address, the other its contents. Be careful not to confuse the two numbers. The address is fixed. It is established when the microcomputer is built. However, the contents may be changed at any time by storing new data.

## Programmed Review

10.	The main function of the _____ is to perform arithmetic or logic operations on the data words that are delivered to it.
11.	(Arithmetic Logic Unit/ALU) The two main inputs to the ALU are called _____. (operands/opcodes)
12.	(operands) The _____ is a temporary storage location for data going to or coming from the data bus.
13.	(data register) The _____ controls the sequence in which the instructions in a program are performed.
14.	(program counter) Each location in memory is assigned a unique number called its _____.
15.	(address) When information is read out of a specific memory location, the contents of that location _____ disturbed. (are/are not)
16.	(are not) During the _____ phase, an instruction is taken from memory and decoded by the MPU. (fetch/execute)
17.	(fetch) The time required to complete the execute phase _____ vary from one instruction to the next. (may/may not)
18.	(may) In an instruction set, the abbreviated version of an instruction is referred to as a _____.
19.	(mnemonic) In an instruction set, the _____ is the binary number that the computer and the programmer use to represent a given instruction. (operand/opcode)
20.	(opcode) The MPU _____ interpret numbers other than binary. (can/cannot)
	(cannot)



## EXECUTING A PROGRAM

Before a program can be run, it must be placed in memory. Later, you will see how this is done. For now, assume that we have already loaded the program developed in the previous section.

The important registers of the microcomputer are shown in Figure 4-12. Notice that the 5-byte program for adding 7 and 10 is shown in memory. The following paragraphs will take you through the step-by-step procedure the computer uses to execute the program.

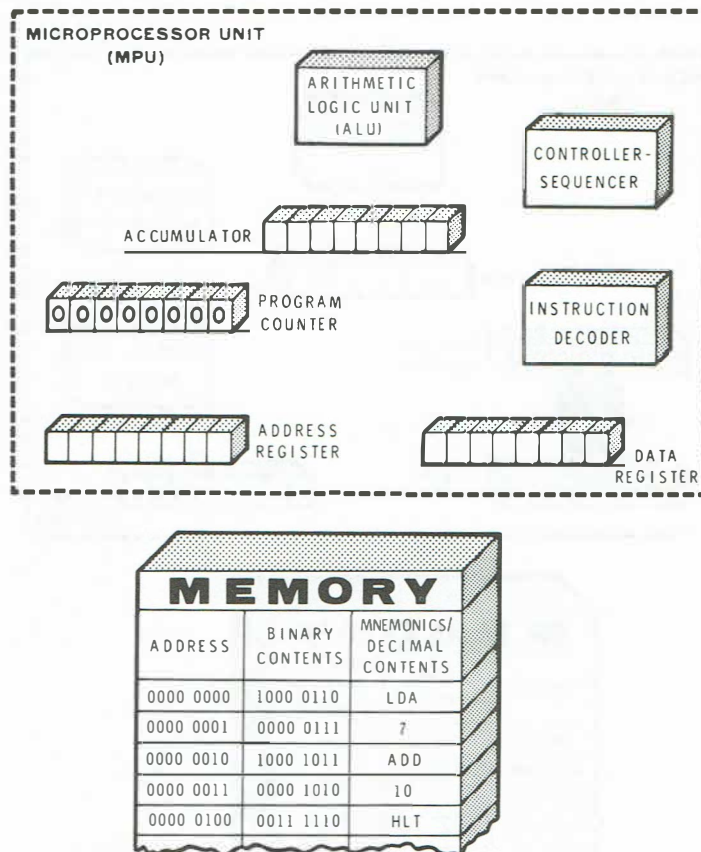


Figure 4-12  
The program counter is set to the address of the first instruction.

To begin executing the program, the program counter must be set to the address of the first instruction. In this case, the first instruction is in memory location 0000 0000, therefore the program counter is set accordingly. The procedure for setting the program counter to the proper address will be discussed later.

## The Fetch Phase

The first step is to fetch the first instruction from memory. The sequence of events that happen during the fetch phase is controlled by the controller-sequencer. It produces a number of control signals which will cause the events illustrated in Figures 4-13 through 4-17 to occur.

First, the contents of the program counter are transferred to the address register, as shown in Figure 4-13. Recall that this is the address of the first instruction.

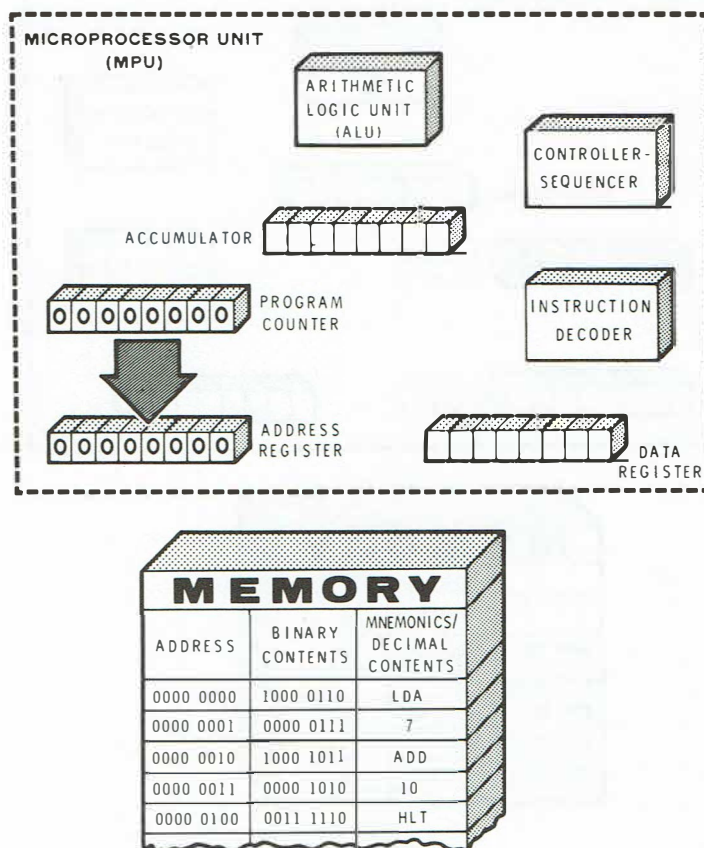


Figure 4-13

The contents of the program counter are transferred to the address register.

Once the address is safely in the address register, the program counter is incremented (increased) by one. That is, its contents change from 0000 0000 to 0000 0001, as shown in Figure 4-14. Notice that this does not change the contents of the address register in any way.

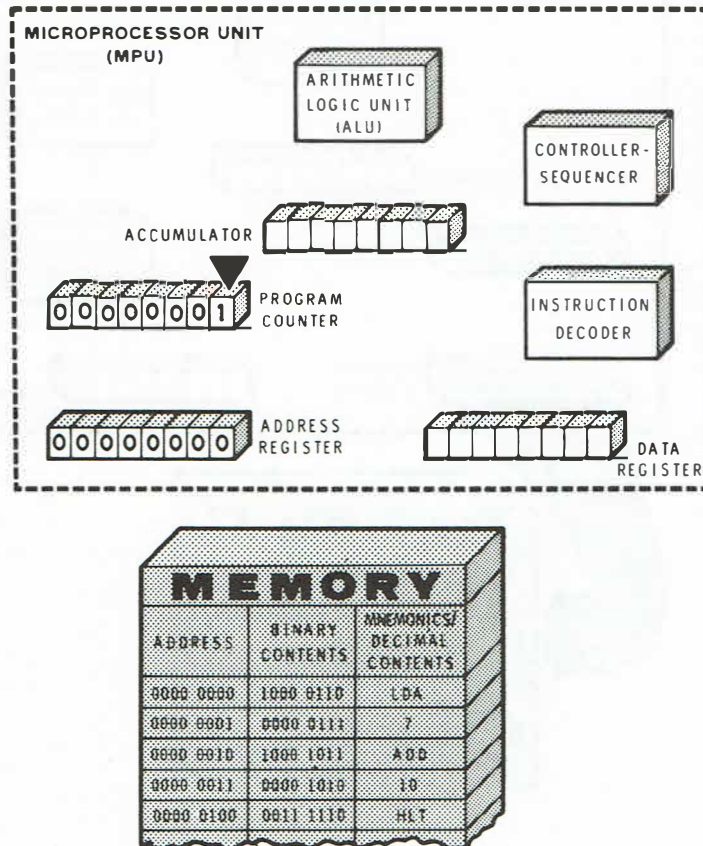


Figure 4-14  
The program counter is incremented.

Next, the contents of the address register (0000 0000) are placed on the address bus (Figure 4-15). The memory circuits decode the address and select memory location 0000 0000.

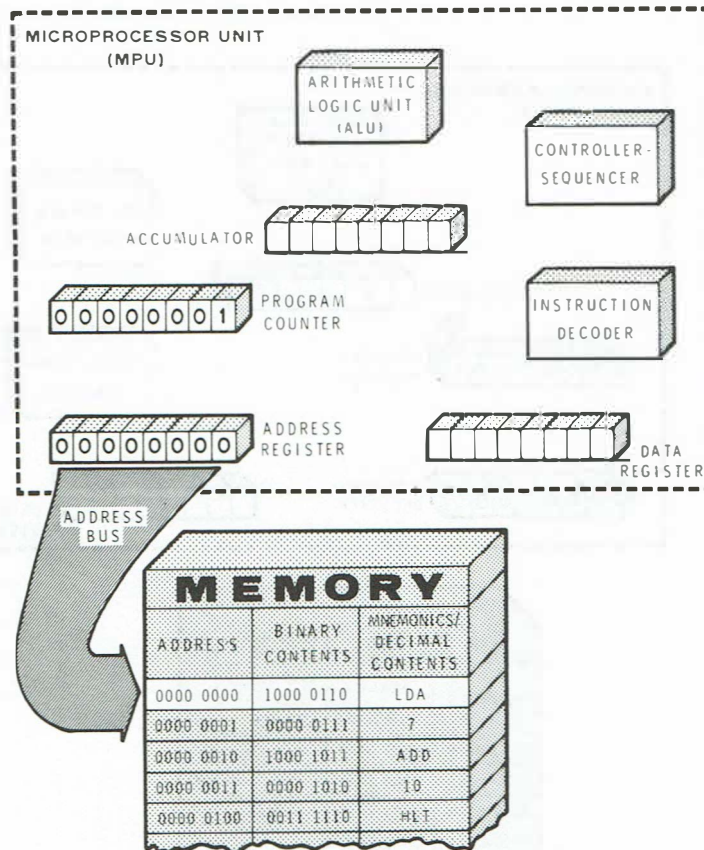


Figure 4-15  
The address of the first instruction is placed on the data bus.

The contents of the selected memory location are placed on the data bus and transferred to the data register in the MPU. After this operation, the opcode for the LDA instruction will be in the data register as shown in Figure 4-16.

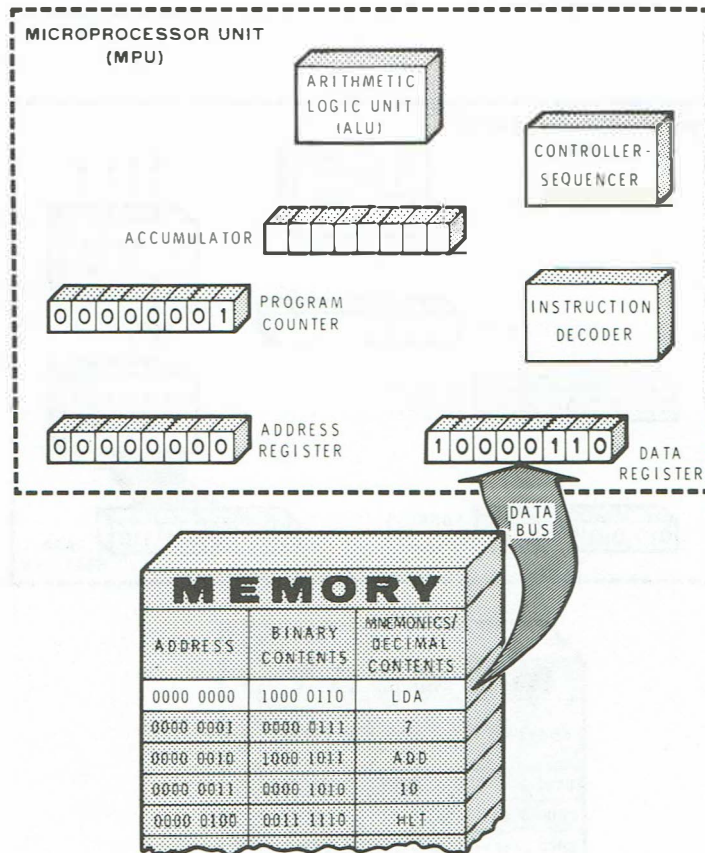


Figure 4-16  
The opcode of the first instruction is placed on the data bus.



The next step is to decode the instruction (Figure 4-17). The opcode is transferred to the instruction decoder, which recognizes that the opcode is an LDA instruction. It informs the controller-sequencer of this fact, and the sequencer produces the necessary control pulses to carry out the instruction. This completes the fetch phase of the first instruction.

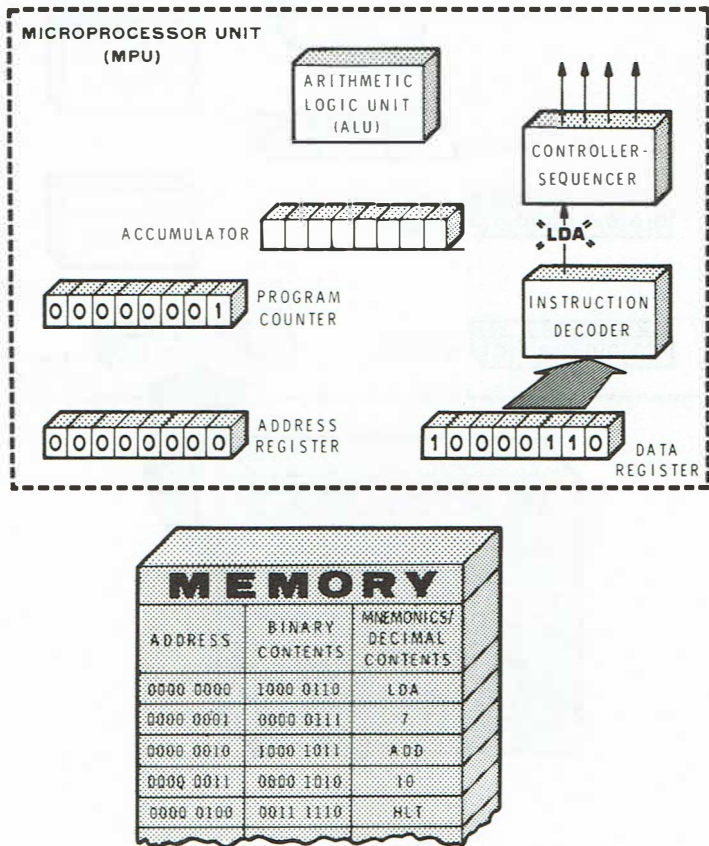


Figure 4-17  
The opcode is decoded.

## The Execute Phase

The first instruction was fetched from memory and decoded during the fetch phase. The MPU now knows that this is an LDA instruction. During the execute phase, it must carry out this instruction by reading out the next byte of memory and placing it in the accumulator.

The first step is to transfer the address of the next byte from the program counter to the address register (Figure 4-18). You will recall that the program counter was incremented to the proper address (0000 0001) during the previous fetch phase.

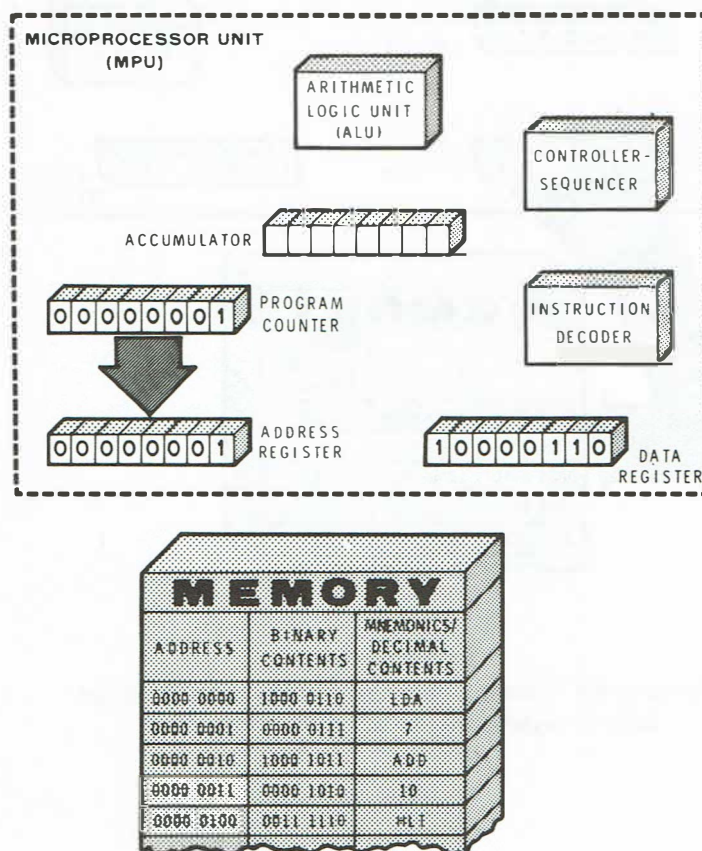


Figure 4-18  
The contents of the program counter are transferred to the address register.

The next two operations are shown in Figure 4-19. First, the program counter is incremented to 0000 0010 in anticipation of the next fetch phase. Second, the contents of the address register (0000 0001) are placed on the address bus.

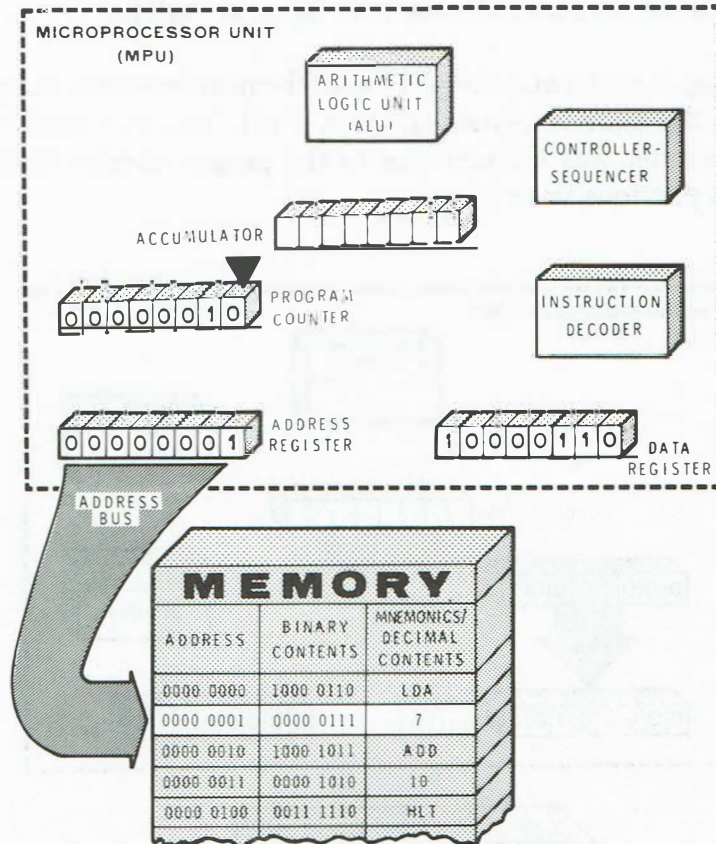


Figure 4-19

The program counter is incremented; the contents of the address register are placed on the address bus.

The address is decoded and the contents of memory location 0000 0001 are loaded into the data register as shown in Figure 4-20. Recall that this is the number 7. An instant later, the number is transferred to the accumulator; thus, the first execute phase ends with the number 7 in the accumulator.

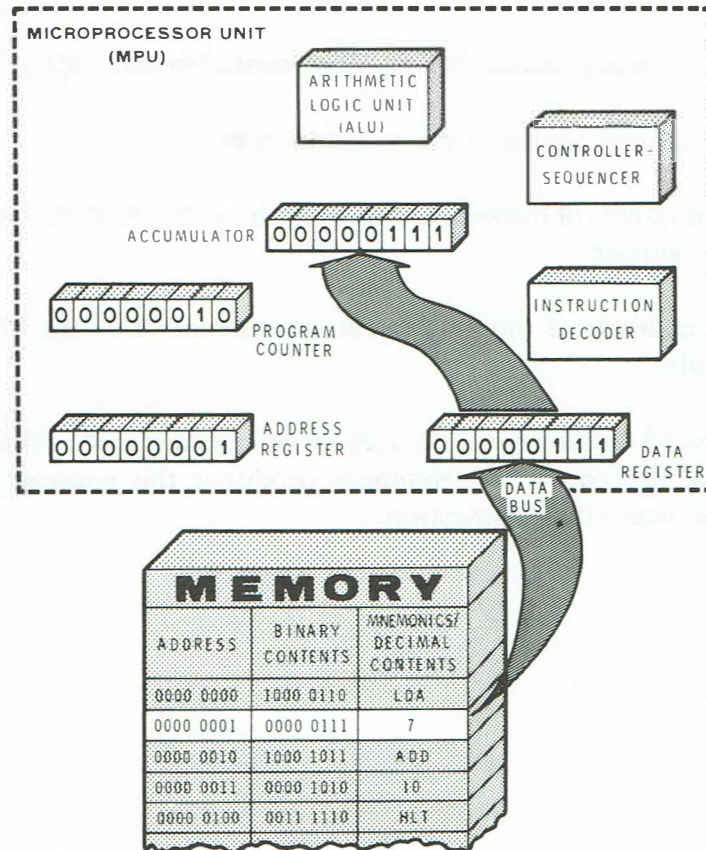


Figure 4-20

The first operand is transferred to the accumulator via the data register.

## Fetching the ADD Instruction

The next instruction in our program is the ADD instruction. It is fetched from memory using the same procedure outlined above. Figure 4-21 illustrates this 5-step procedure.

1. The contents of the program counter (0000 0010) are transferred to the address register.
2. The program counter is then incremented to 0000 0011.
3. The address is placed on the address bus.
4. The contents of the selected memory location are transferred to the data register.
5. The contents of the data register are decoded by the instruction decoder.

The data word fetched from memory is the opcode for the ADD instruction. Thus, the controller-sequencer produces the necessary control pulses to execute this instruction.



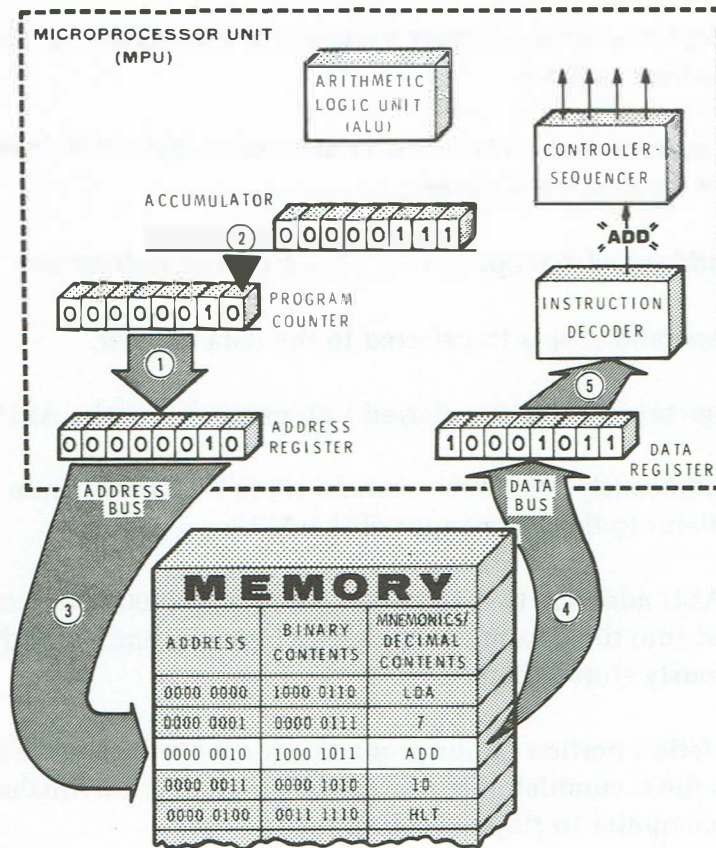


Figure 4-21  
Fetching the ADD instruction.

## Executing the ADD Instruction

The execution of the ADD instruction is a 6-step procedure. This procedure is illustrated in Figure 4-22.

1. The contents of the program counter (0000 0011) are transferred to the address register.
2. The program counter is then incremented to 0000 0100 in anticipation of the next fetch phase.
3. The address of the operand is placed on the address bus.
4. The operand (10) is transferred to the data register.
- 5A. The operand (10) is transferred into one input of the ALU.
- 5B. Simultaneously, the other operand (7) is transferred from the accumulator to the other input of the ALU.
6. The ALU adds the two operands. Their sum (0001 0001 or 17) is loaded into the accumulator, destroying the number (7) that was previously stored there.

The computation portion of our program ends with the sum of the two operands in the accumulator. However, the program is not finished until it tells the computer to stop executing instructions.

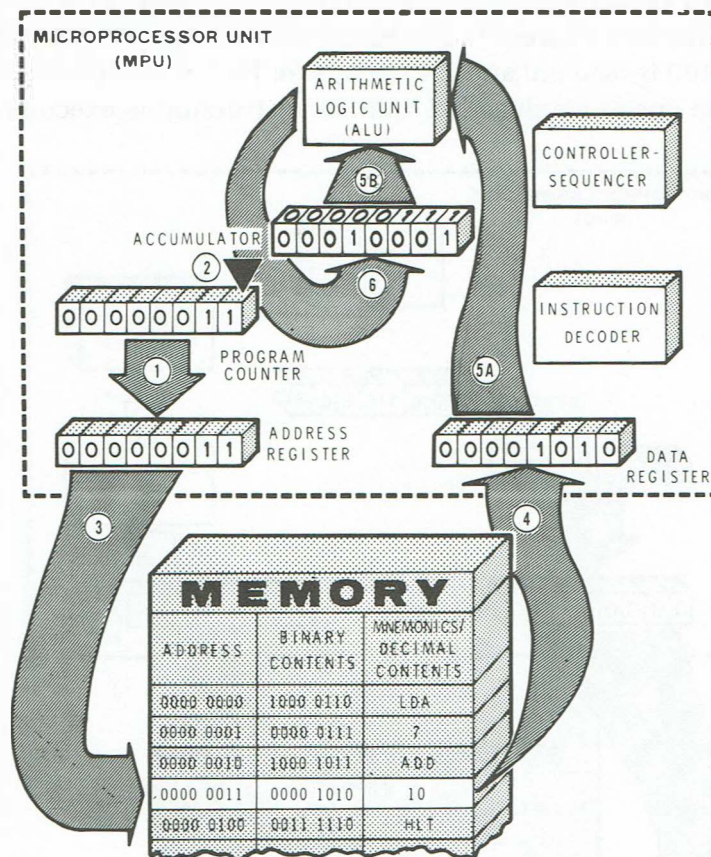


Figure 4-22  
Executing the ADD instruction.

## Fetching and Executing the HLT Instruction

The final instruction in the program is a HLT instruction. It is fetched using the same fetch procedure as before. The five steps are illustrated in Figure 4-23. The address comes from the program counter via the address register. When this address is placed on the address bus, memory location 0000 0100 is read out and the opcode for HLT is loaded into the data register. The opcode is decoded and the instruction is executed.

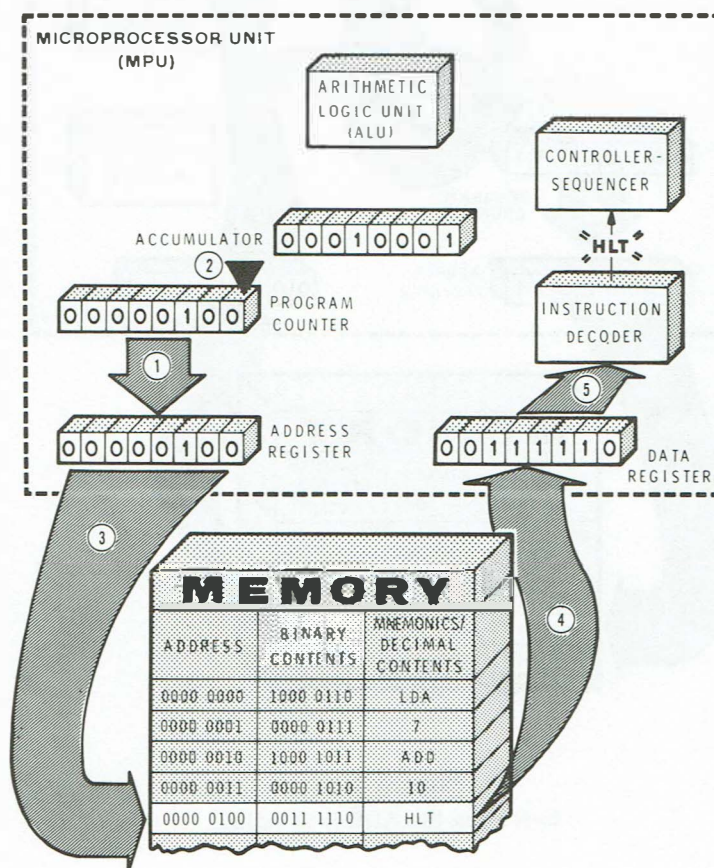


Figure 4-23  
Fetching and executing the HLT instruction.

The execution of the HLT instruction is very simple. The controller-sequencer simply stops producing control signals. Consequently, all computer operations stop. Notice that the program has accomplished our objective of adding 7 to 10. The resulting sum, 17, is now in the accumulator.



## Programmed Review

21.	When first executing a program, the program counter must be set to the _____ of the first instruction.
22.	(address) When you increment the program counter, you _____ the count by one. (increase/decrease)
23.	(increase) During the fetch phase of program execution, the contents of the address register must first be placed on the _____ before they can be decoded. (address bus/data bus)
24.	(address bus) During program execution, the _____ produces the necessary control pulses to execute the instructions.
25.	(controller-sequencer) The instruction which tells the computer to stop executing instructions is called the _____ instruction.
26.	(halt or HLT) Upon completion of the program shown in Figure 4-10, the results of the program are stored in the _____.
	(accumulator)



## ADDRESSING MODES

If you examine the program discussed in the previous section, you will find that it uses two distinctly different types of instructions. One type of instruction requires an operand. LDA and ADD are examples of this type. These are two-byte instructions. The first byte is the opcode; the second is the operand.

Microprocessors also have single-byte instructions. HLT is a good example. This instruction requires no operand; thus, it can be implemented with a single byte.

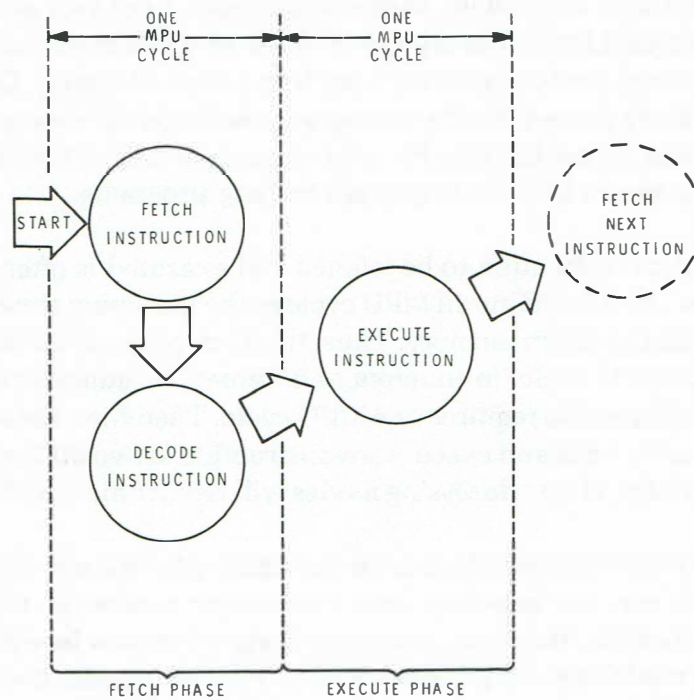
Instructions can be classified in several different ways. One of the most basic distinctions is their addressing mode. The addressing mode refers to the method by which an instruction addresses its operand.

### Inherent or Implied Addressing

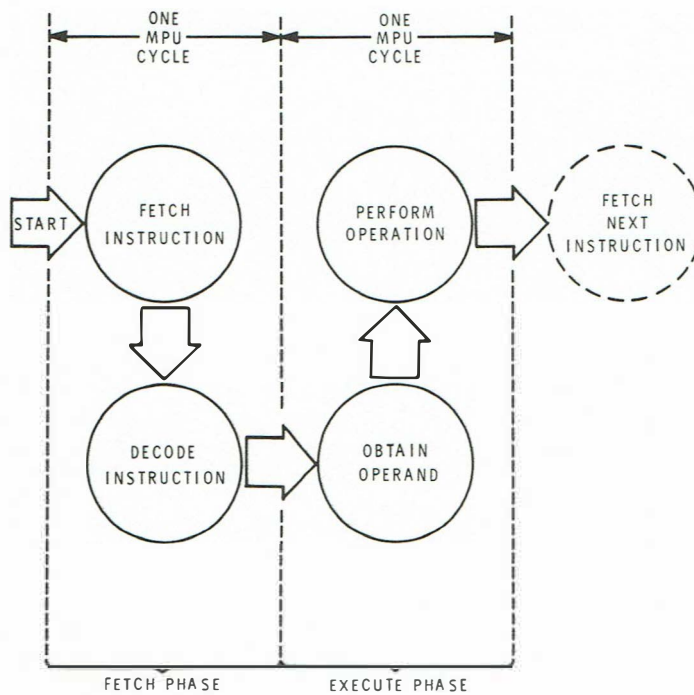
Single-byte instructions have no operand, or the operand is implied by the opcode itself. As an example, the HLT instruction had no operand at all. However, some single-byte instructions may have an implied operand, for example, the “increment accumulator” instruction. The number that is operated upon (incremented) is the number in the accumulator. This instruction simply adds one to the contents of the accumulator. This type of addressing will be referred to in this course as either **implied addressing** or **inherent addressing**. The operations performed during an instruction of this type are illustrated in Figure 4-24.

### Immediate Addressing

In our previous program, the 2-byte instructions used the **immediate addressing** mode. In this mode, the operand is the byte immediately following the opcode. That is, the byte of data that is to be operated upon is the second byte of the instruction. When these 2-byte instructions are stored in memory, the address of the operand is the memory location following the opcode. The operations performed during this type of instruction are illustrated in Figure 4-25.



**Figure 4-24**  
Operations performed in the inherent  
or implied addressing mode.



**Figure 4-25**  
Operations performed in the immediate addressing mode.

The inherent and immediate addressing modes have two advantages. First, they require little memory space — one and two bytes respectively — which is important because memory locations cost money. Generally, the less memory space taken by a program, the better off we are. Second, these addressing modes require a minimum of execution time. The execution time can become important in long programs.

The time for an instruction to be fetched and executed is often given in **MPU cycles**. We will define an MPU cycle as the minimum time required to fetch a data byte from memory. Thus, the fetch phase of an instruction requires one MPU cycle. In inherent and immediate addressing modes, the execute phase also requires one MPU cycle. Therefore, the minimum time required to fetch and execute any instruction is two MPU cycles. As you will see later, other addressing modes will require more MPU cycles.

Because of their fast execution time and their efficient use of memory, you should use the inherent and immediate modes of addressing whenever possible. However, there are many situations in which these addressing modes are simply not suitable. For this reason, every microprocessor will have instructions that use other addressing modes.

## Direct Addressing

Most computer operations involve an operand. To realize its full potential, the computer must be able to manipulate the operand in many different ways. Immediate addressing of an operand is most useful when the operand is a constant that is used by only one instruction in the program. In this case, the operand can be placed immediately after that instruction's opcode.

However, there are many cases in which the operand is a variable, as is the case in many robot control operations, which may be operated upon by many different instructions. In these cases, the immediate addressing mode is simply not practical, and a more sophisticated form of addressing is necessary.

One way of solving this problem is to use the **direct addressing mode**. In this mode, an instruction requires two bytes of memory. The first byte is the opcode of the instruction, just as before. However, the second byte is **not** the operand. Instead, the second byte is the **address** of the operand. The format of the instruction as it appears in memory is shown in Figure 4-26.

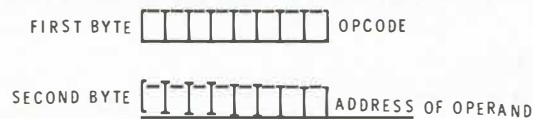


Figure 4-26  
Format of an instruction which  
uses the direct addressing mode.

Three typical direct-addressing-mode instructions are listed in Figure 4-27. The first is the load accumulator instruction (LDA). Read the description carefully and note the difference between this instruction and the LDA immediate instruction discussed earlier. An example of each may help. In the **immediate addressing mode**, the instruction

LDA 50<sub>10</sub>

means "load 50<sub>10</sub> into the accumulator." But in the **direct addressing mode**, the same instruction means "load the number at memory location 50<sub>10</sub> into the accumulator."

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	1001 0110 <sub>2</sub> or 96 <sub>16</sub>	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add	ADD	1001 1011 <sub>2</sub> or 9B <sub>16</sub>	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator	STA	1001 0111 <sub>2</sub> or 97 <sub>16</sub>	Store the contents of the accumulator in the memory location whose address is given by the next byte.

Figure 4-27

Direct addressing mode instructions.

You may have noticed that this instruction has a different opcode from the LDA immediate instruction. This is necessary to tell the MPU the exact nature of the instruction. That is, the opcode tells the MPU the addressing mode as well as the operation that is to be performed.

The ADD instruction also has a slightly different meaning in the direct addressing mode. Recall that, in the **immediate** addressing mode,

ADD 10<sub>10</sub>

means “add 10<sub>10</sub> to the contents of the accumulator.” However, in the direct addressing mode,

ADD 10<sub>10</sub>

means “add the contents of memory location 10<sub>10</sub> to the contents of the accumulator.” Once again, the opcode tells the MPU the addressing mode. An opcode of 8B<sub>16</sub> means ADD immediate whereas an opcode of 9B<sub>16</sub> means add direct.

The last instruction shown is a store accumulator (STA) instruction. It tells the MPU to store the contents of the accumulator in the address indicated by the second byte of the instruction.



For example:

STA 20<sub>10</sub>

means “store the contents of the accumulator in memory location 20<sub>10</sub>.” Because the second byte of the instruction must be an address, there is no STA immediate instruction.

The direct addressing mode instructions require one or more additional MPU cycles to execute. A typical fetch-execute sequence is illustrated in Figure 4-28. The instruction fetch is the same regardless of the addressing mode. However, the execution phase is extended since the MPU must first obtain the address of the operand from memory and then retrieve the operand itself from memory.

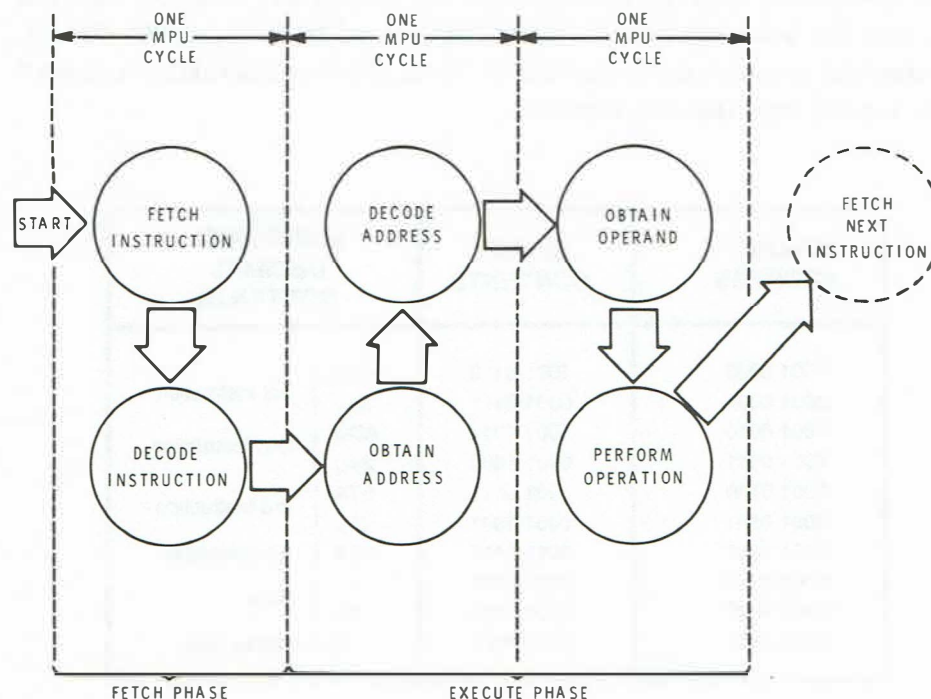


Figure 4-28

Most direct addressing mode instructions require three MPU cycles.

Direct addressing generally requires more memory and longer execution times. However, there are many cases in which the added flexibility makes direct addressing worthwhile in spite of these disadvantages.

Sample Program Using Direct Addressing

The difference between direct and immediate addressing can best be illustrated by a sample program. Earlier we examined a program that added two numbers, 7 and 10, and placed the sum in the accumulator. Now let's look at the same program using direct addressing, only this time, the sum will be stored in memory.

Figure 4-29 shows the program as it would look when stored in memory. Assume that we have arbitrarily stored the program starting at memory location or address 0001 0000<sub>2</sub> (16<sub>10</sub>). Addresses 16<sub>10</sub> and 17<sub>10</sub> contain the first instruction:

```
LDA 2310
```

This instruction tells the MPU to load the contents of memory location 23<sub>10</sub> into the accumulator. Looking down to address 23<sub>10</sub> (0001 0111<sub>2</sub>), you see that it contains the operand 7. Thus, the first instruction causes 7 to be loaded into the accumulator.

BINARY ADDRESS	BINARY CONTENTS	MNEMONIC/ DECIMAL CONTENTS
0001 0000	1001 0110	LDA
0001 0001	0001 0111	23 <sub>10</sub>
0001 0010	1001 1011	ADD
0001 0011	0001 1000	24 <sub>10</sub>
0001 0100	1001 0111	STA
0001 0101	0001 1001	25 <sub>10</sub>
0001 0110	0011 1110	HLT
0001 0111	0000 0111	7
0001 1000	0000 1010	10
0001 1001	0000 0000	Reserved for sum

Figure 4-29  
Sample program using direct addressing.

The second instruction is in memory locations  $18_{10}$  and  $19_{10}$  and is:

ADD  $24_{10}$

This tells the MPU to add the number at address  $24_{10}$  to the number in the accumulator. Address  $24_{10}$  ( $0001\ 1000_2$ ) contains the number  $10_{10}$ . Therefore, the second instruction causes  $10_{10}$  to be added to the contents of the accumulator. The sum ( $17_{10}$ ) is placed back in the accumulator.

The third instruction, in location  $20_{10}$  and  $21_{10}$  is:

STA  $25_{10}$

This tells the MPU to store the contents of the accumulator in memory location  $25_{10}$ . After this instruction is executed, the number  $17_{10}$  will appear in location  $25_{10}$ .

The final instruction tells the MPU to halt operation. The program illustrates the value of the HLT instruction. Let's assume that the HLT instruction is inadvertently omitted. In this case, the MPU would fetch the next byte in sequence and attempt to execute it as if it were an instruction. The next byte is the number  $7_{10}$ . This is a data word and was never intended to be an instruction. Nevertheless, the MPU probably has an instruction with an opcode of  $7_{10}$ . After executing this instruction, the MPU will continue to fetch and execute whatever it finds in the remaining memory locations. Without a HLT instruction, it has no way of knowing where the instructions end and data begins.

## Executing the Sample Program

The data flow within the microcomputer is slightly different for the direct addressing mode. Figure 4-30 shows several of the data paths within the microcomputer. Using this type of diagram, let's examine the data manipulations that occur during the execution of our sample program.

Notice that our program is loaded in memory starting at address  $16_{10}$ . The program counter is set to  $16_{10}$ , so the MPU is ready to begin executing the program.

The first fetch phase is illustrated in Figure 4-30, and during this phase:

1. The contents of the program counter are loaded into the address register.
2. The program counter is then incremented to  $17_{10}$ , (00010001).
3. The contents of the address register are placed on the address bus.
4. The contents of the selected memory location are transferred via the data bus to the data register.
5. The contents of the data register are decoded.
6. The MPU recognizes that an LDA direct operation is indicated. This concludes the fetch phase.

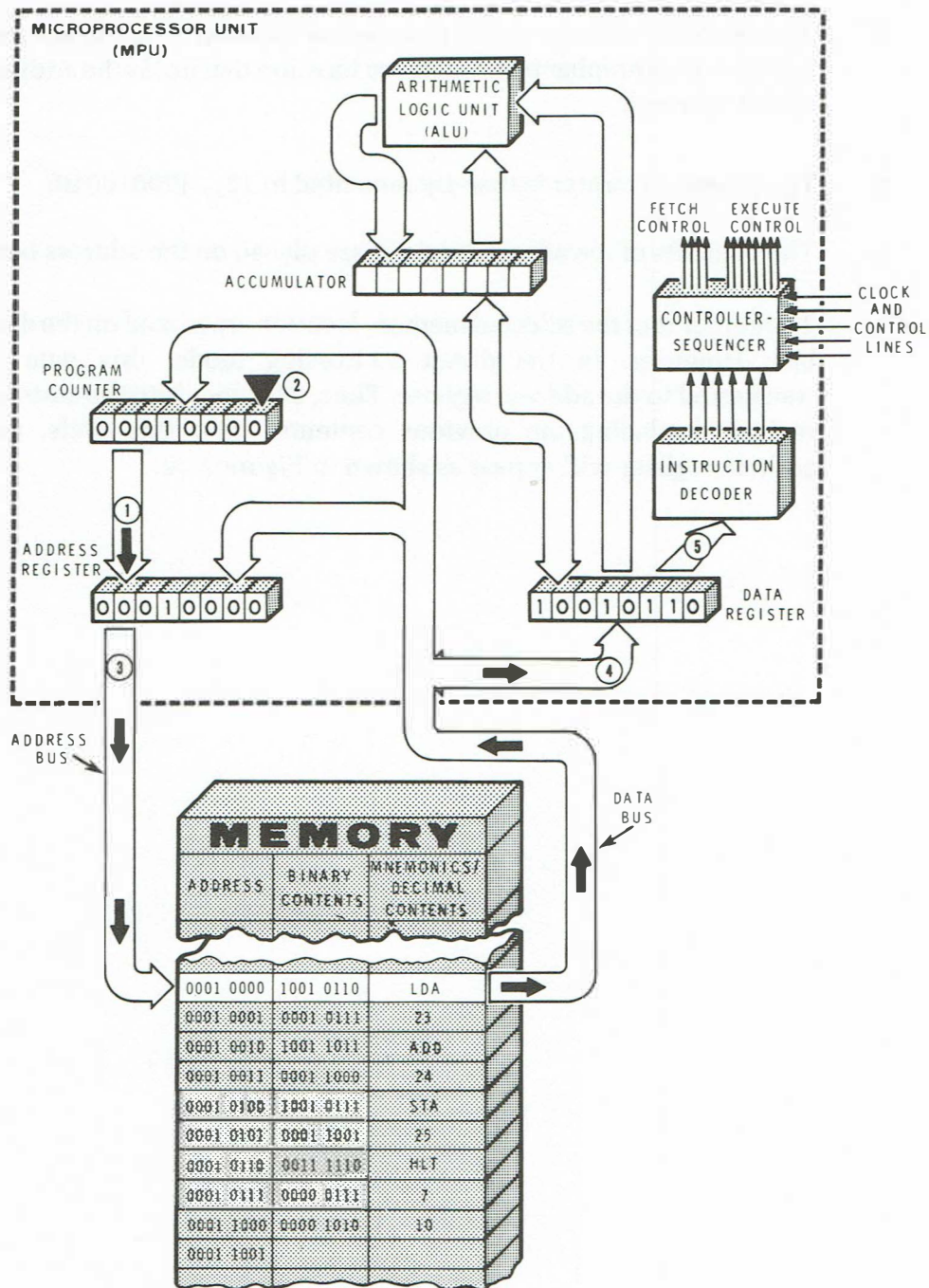


Figure 4-30  
Fetching the opcode of the first instruction.



The execute phase has two parts when direct addressing is indicated. Figure 4-31 illustrates the first half of the execute phase. During this half:

1. The contents of the program counter are transferred to the address register. This number is the memory location that holds the address of the operand.
2. The program counter is then incremented to  $18_{10}$ , (00010010).
3. The contents of the address register are placed on the address bus.
4. The contents of the selected memory location are placed on the data bus. However, in the direct addressing mode, this data is transferred to the address register. Thus,  $23_{10}$  goes into the address register, replacing the previous contents. After this cycle, the address register will appear as shown in Figure 2-32.

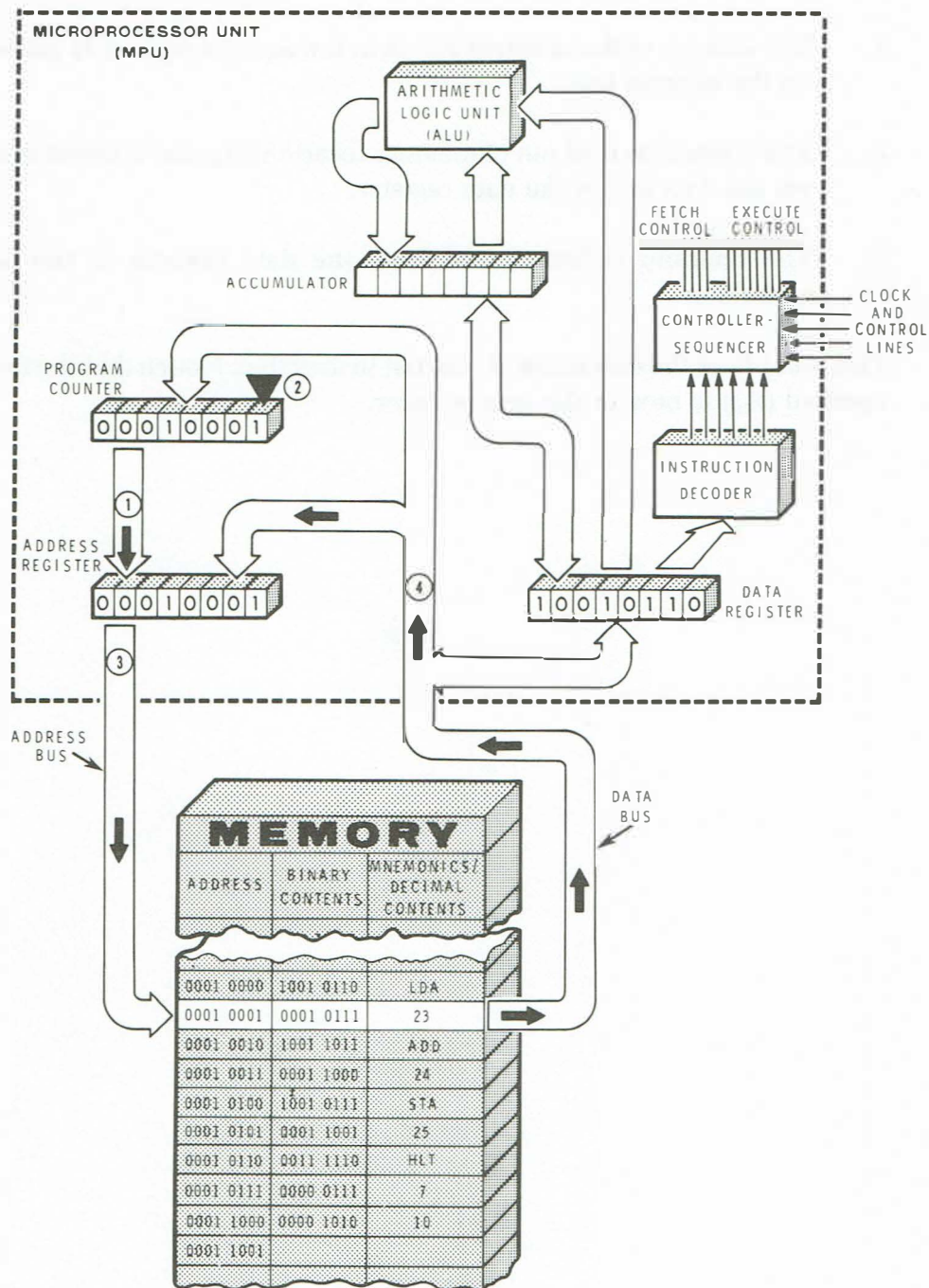


Figure 4-31  
Fetching the address of the first operand.

During the second half of the execute phase, the operand is loaded into the accumulator as shown in Figure 4-32. The procedure is:

1. The address of the operand that is in the address register is placed on the address bus.
2. The operand is read out of memory location  $23_{10}$  and is transferred via the data bus to the data register.
3. The operand is transferred from the data register to the accumulator.

This completes the execution of the first instruction. Notice that the first operand ( $7_{10}$ ) is now in the accumulator.

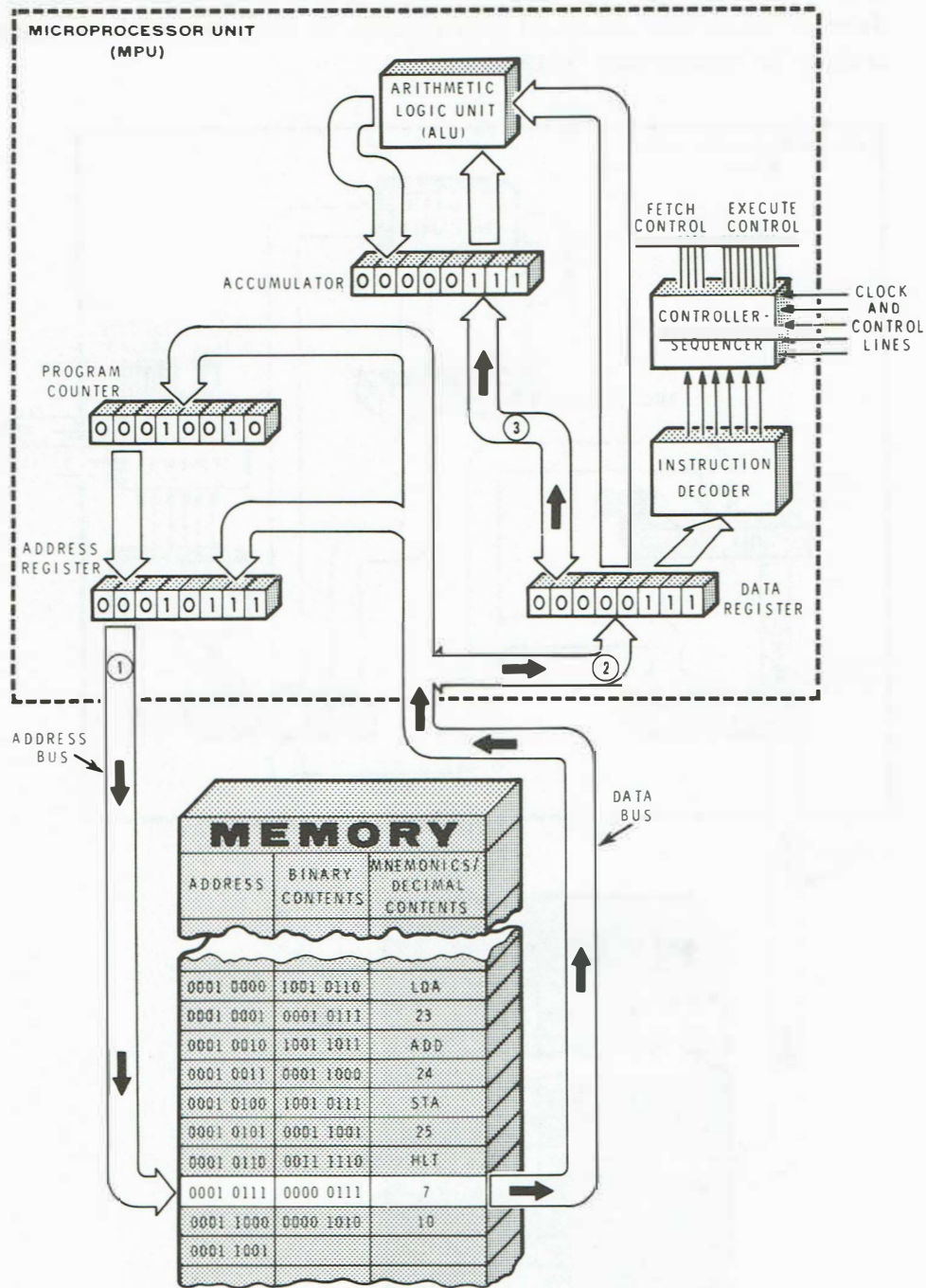


Figure 4-32  
Fetching the first operand.

The fetch phase for the second instruction is similar to that of the first. As shown in Figure 4-33, it causes the opcode of the ADD instruction to be read out of address 18<sub>10</sub>. The opcode is transferred to the instruction decoder via the data bus and data register. In the process, the program counter is incremented to 19<sub>10</sub>.

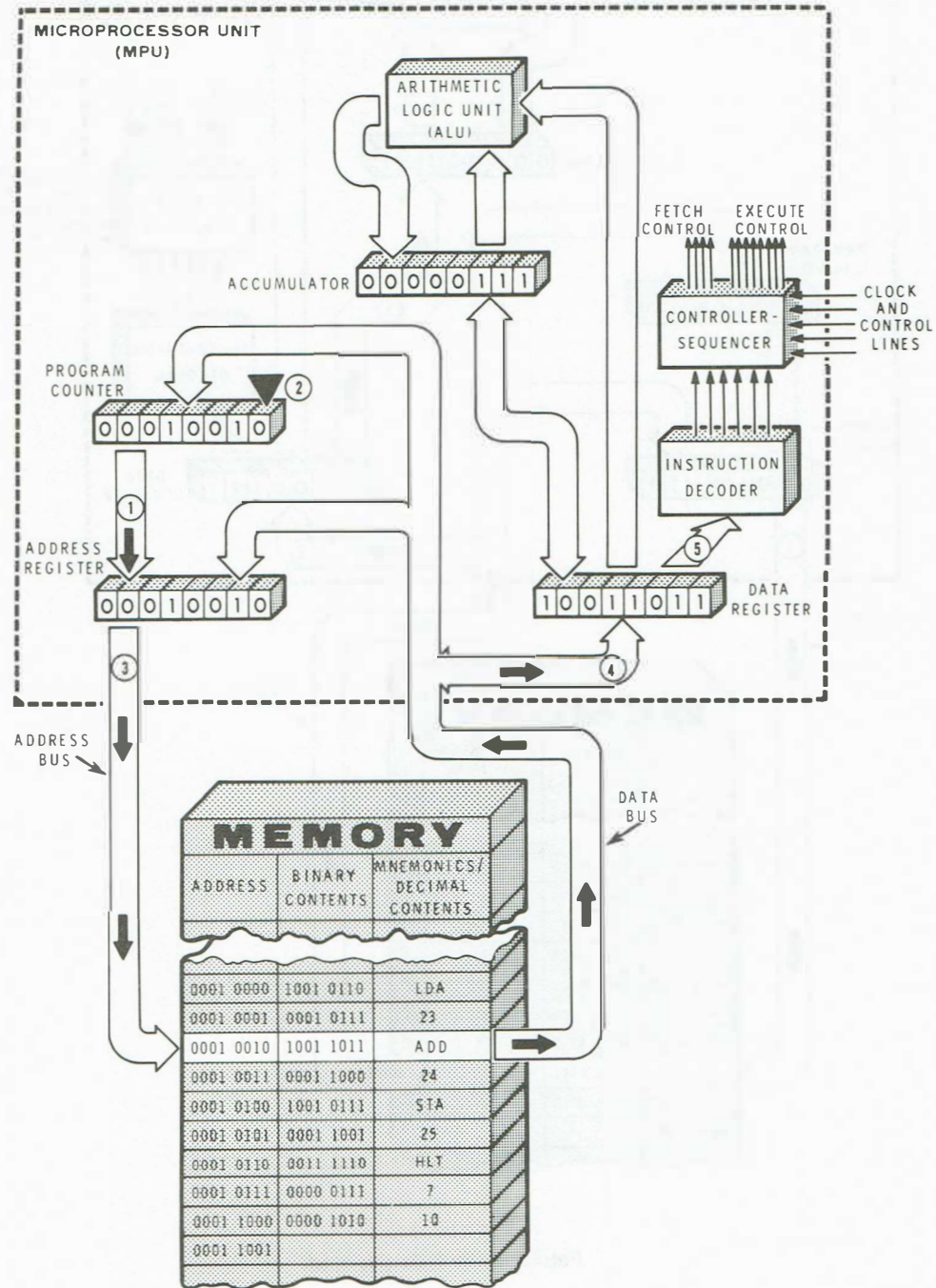


Figure 4-33  
Fetching the opcode of the second instruction.



The first half of the execute phase is illustrated in Figure 4-34. Here, the address of the second operand is read out of memory location 19<sub>10</sub> and is placed in the address register.

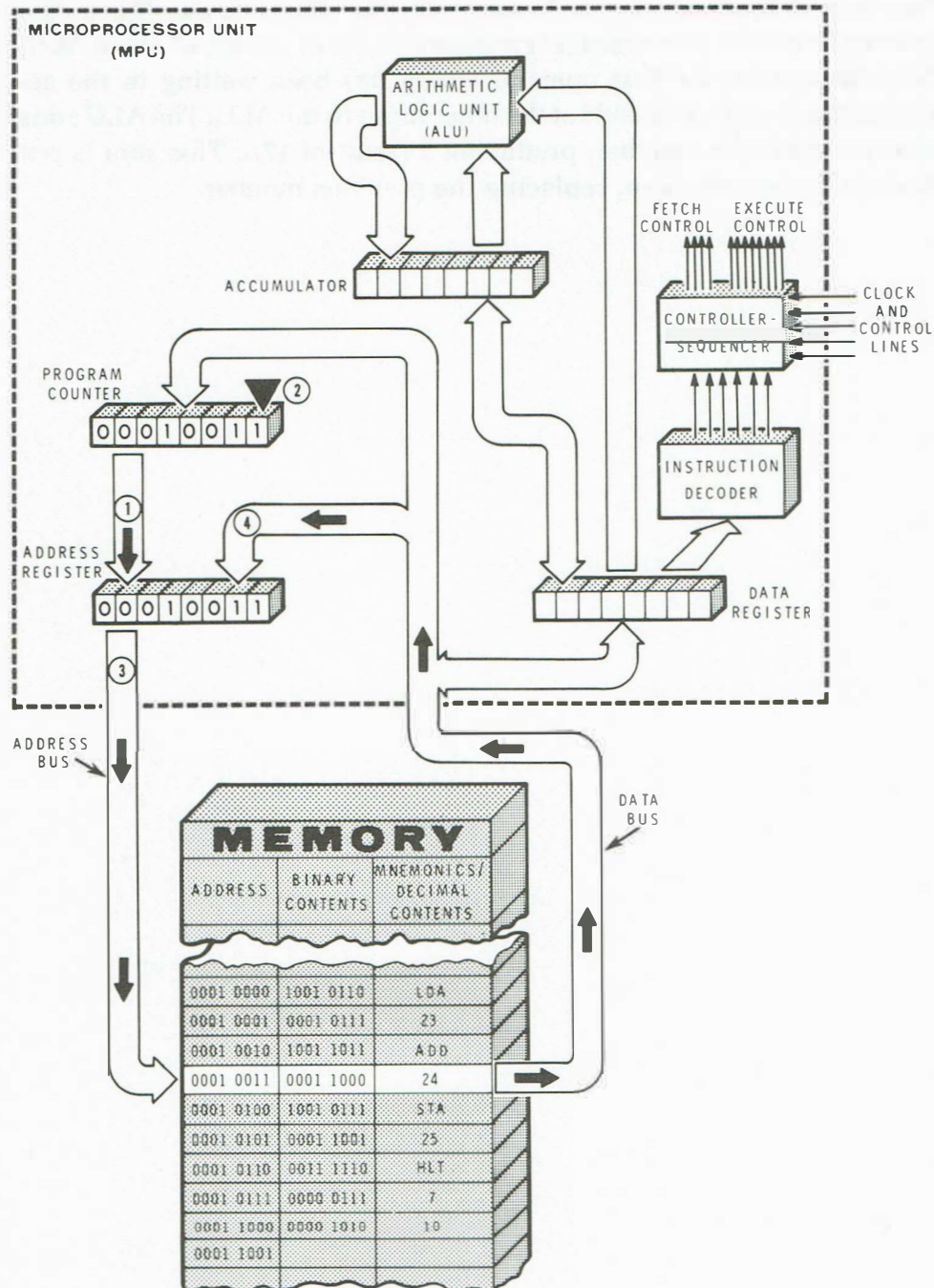


Figure 4-34  
Fetching the address of the second operand.

Figure 4-35 illustrates the second cycle of the execute phase. Here, the address of the second operand is transferred from the address register to the address bus. The address is  $24_{10}$ . Therefore, the contents of location  $24_{10}$  are placed on the data bus and transferred to the data register. That is, the second operand  $10_{10}$  is loaded into the data register. Then, the operand from the data register is made available at one input of the ALU. Simultaneously, the first operand which has been waiting in the accumulator is made available at the other input to the ALU. The ALU adds the two operands together, producing a result of  $17_{10}$ . This sum is put back in the accumulator, replacing the previous number.

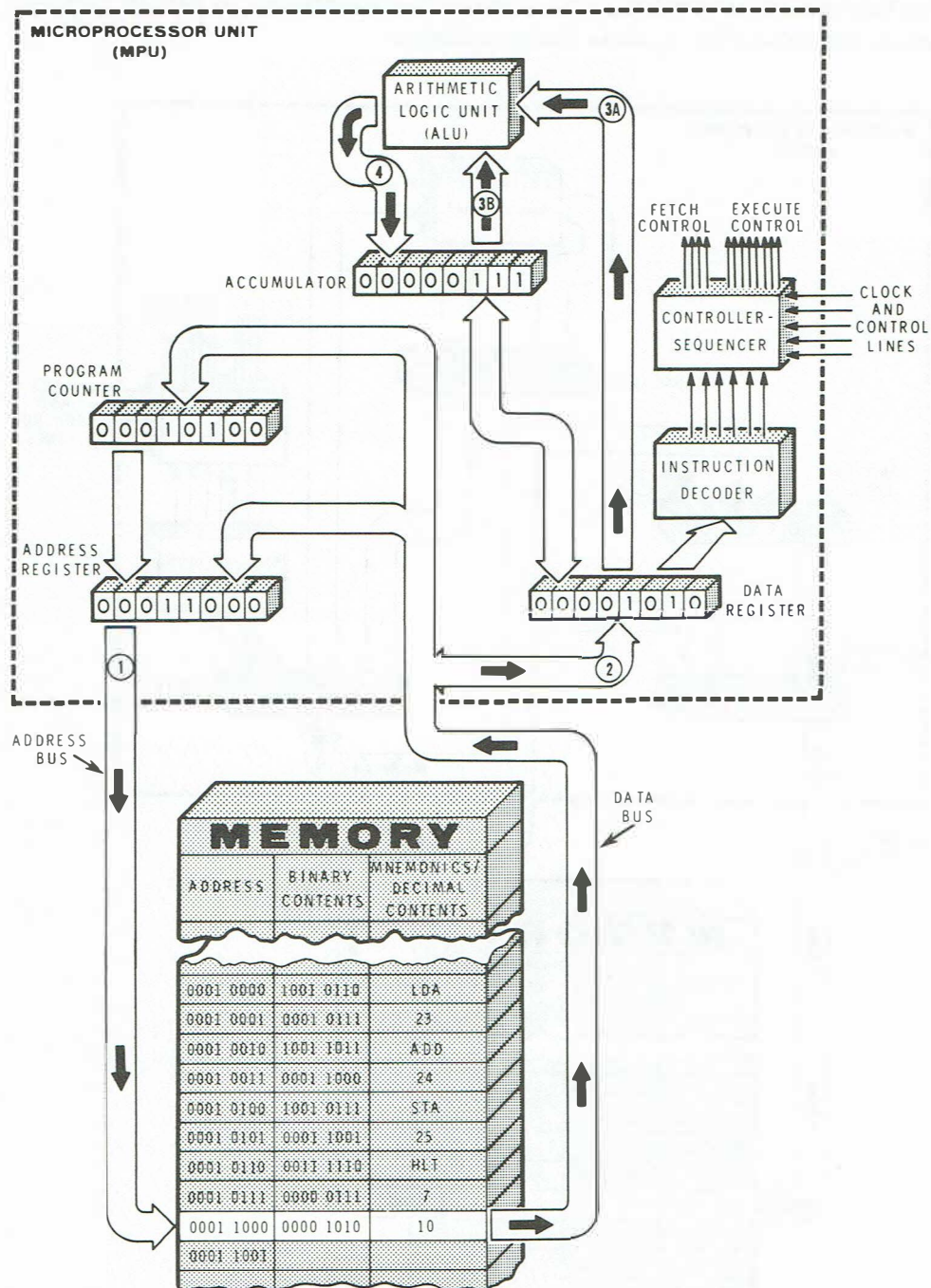


Figure 4-35  
Adding the two operands.

Now all that remains is to place the sum in memory. This is done by the STA 25<sub>10</sub> instruction. Since this is the next instruction in sequence, it will be fetched and executed next. The fetch is illustrated in Figure 4-36. It ends with the STA opcode being decoded.

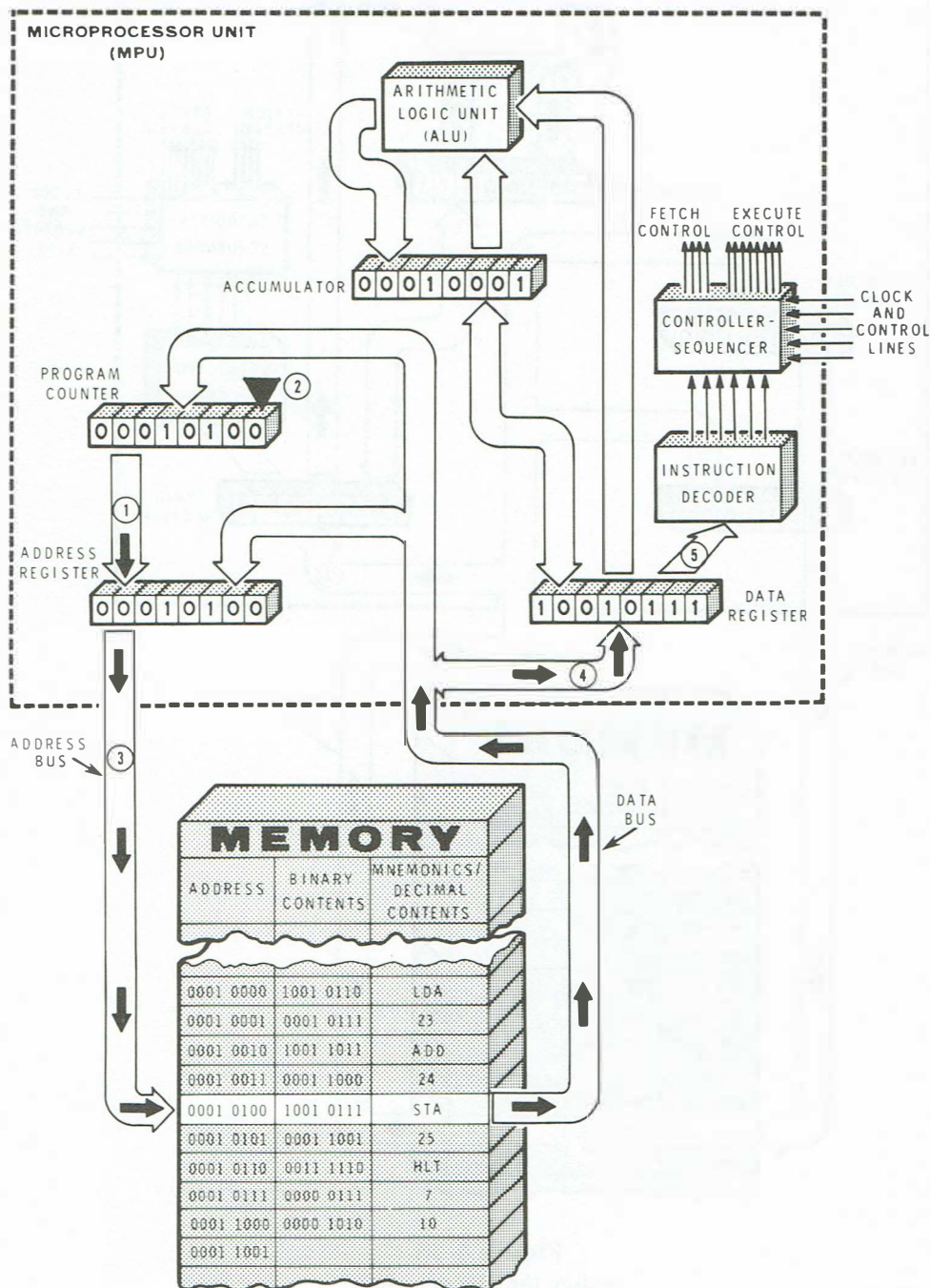


Figure 4-36  
Fetching the third opcode.



The first half of the execution phase of the STA instruction involves loading the address of the storage location into the address register. Figure 4-37 illustrates that this 4-step procedure is identical to that performed for the previous two instructions. It ends with the address 25<sub>10</sub> in the address register.

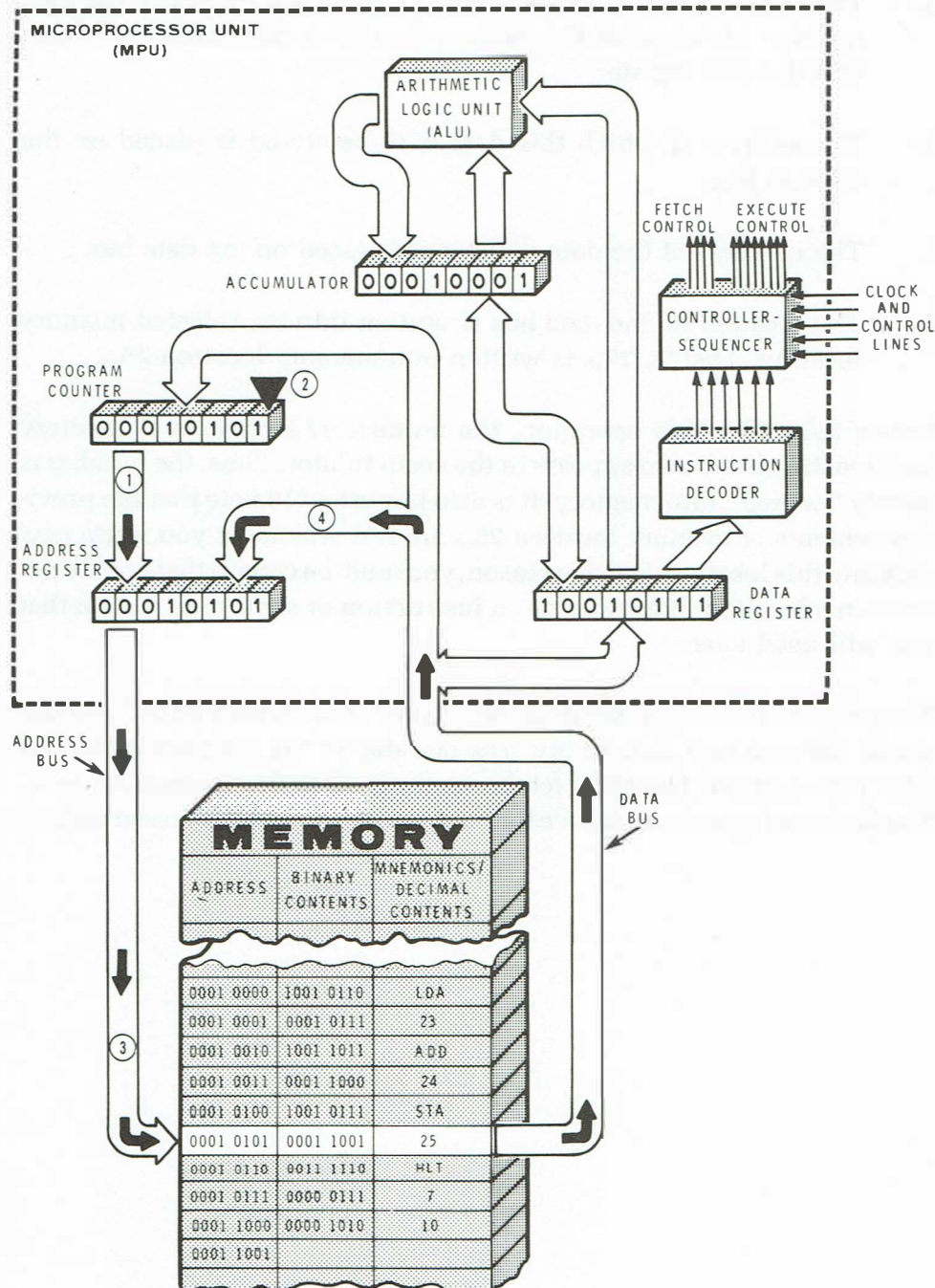


Figure 4-37  
Fetching the third address.



During the final half of the execute phase, the contents of the accumulator are transferred to the data register and are then stored in the selected memory location. We have not yet discussed this operation in detail; therefore, the step-by-step procedure is presented below. Refer to Figure 4-38 for the following steps:

1. The contents of the accumulator ( $17_{10}$ ) are transferred to the data register. At this point, the number 17 exists in both the accumulator and the data register.
2. The address at which this data is to be stored is placed on the address bus.
3. The contents of the data register are placed on the data bus.
4. The location on the data bus is written into the selected memory location. That is,  $17_{10}$  is written into memory location  $25_{10}$ .

Notice that, after this operation, the number  $17_{10}$  appears at memory location  $25_{10}$ , but it also appears in the accumulator. Thus, the number is merely “copied” into memory. It is also important to note that the previous contents of memory location  $25_{10}$  are lost whenever you write new data into this location. For this reason, you must be certain that you do not write in a location that contains an instruction or some byte of data that you will need later.

The program has now accomplished its goal. It has added 10 to 7 and has stored the sum back in memory. The last step in the program is the halt (HLT) instruction. The MPU fetches and executes this instruction next. The fetch and execute sequence for this instruction was discussed earlier.

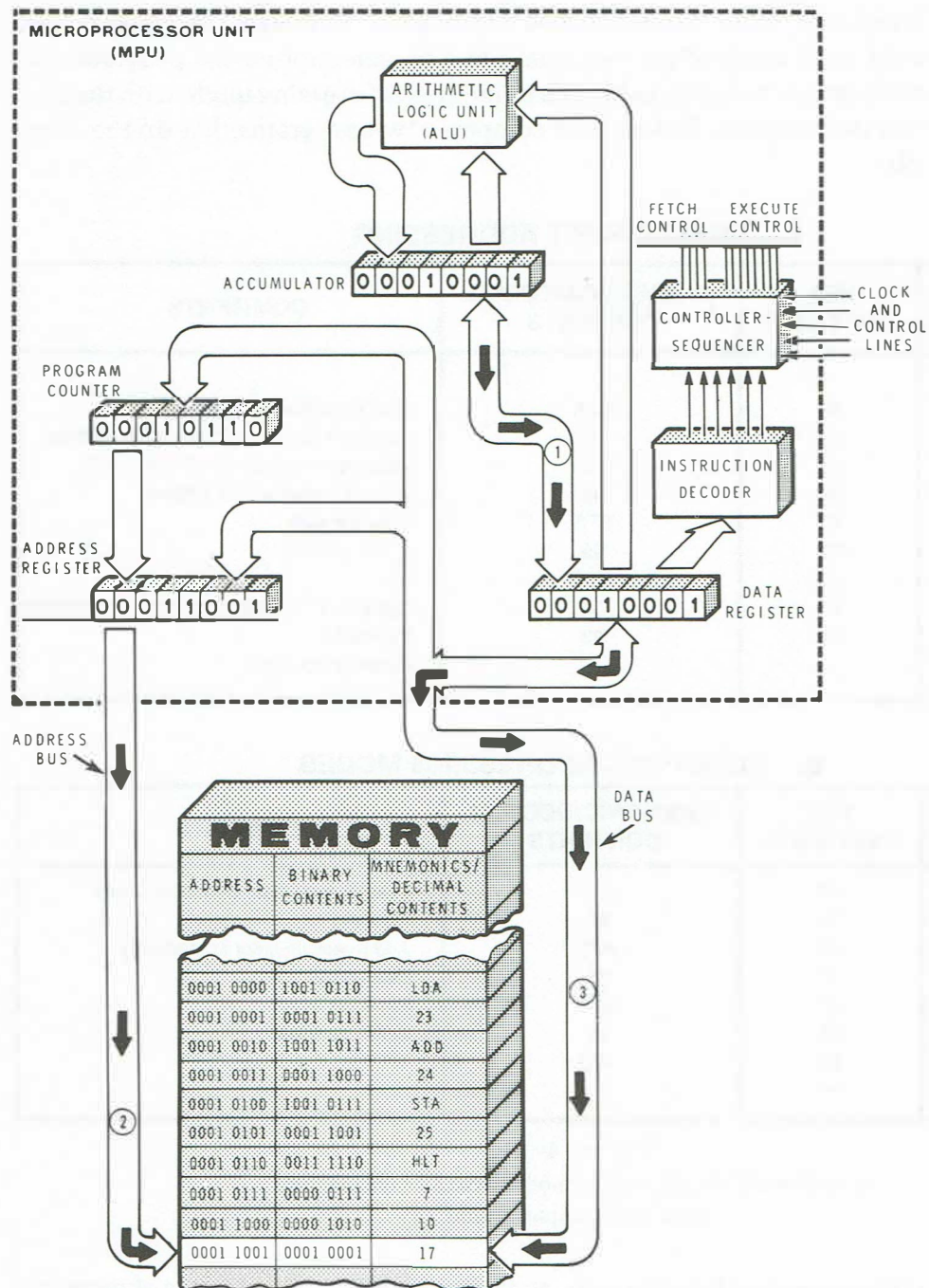


Figure 4-38  
Storing the sum.

## Combining Addressing Modes

When you write programs, you can use the addressing mode that best suits your application. For example, you can shorten the program that was just discussed by using the immediate addressing mode with the first two instructions. Figure 4-39 compares two programs that do the same job.

### A. USING DIRECT ADDRESSING

HEX ADDRESS	HEX CONTENTS	MNEMONIC/DECIMAL CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with operand 1 which is stored at this address.
01	07	07	
02	9B	ADD	Add to accumulator direct operand 2 which is stored at this address.
03	08	08	
04	97	STA	Store the sum
05	09	09	at this address.
06	3E	HLT	Stop
07	21	33	Operand 1
08	17	23	Operand 2
09	—	—	Reserved for sum.

### B. COMBINING ADDRESSING MODES

HEX ADDRESS	HEX CONTENTS	MNEMONIC/DECIMAL CONTENTS	COMMENTS
00	86	LDA	Load accumulator immediately with Operand 1.
01	21	33 <sub>10</sub>	
02	8B	ADD	Add to accumulator immediately Operand 2.
03	17	23 <sub>10</sub>	
04	97	STA	Store the sum
05	07	07	at this address.
06	3E	HLT	
07	—	—	

Figure 4-39

By combining the addressing modes, we can save memory space and computer time.

Using direct addressing only, the program required ten bytes of memory, and its execution requires eleven MPU cycles. However, if you use immediate addressing for the first two instructions, the program requires only eight bytes of memory and can be executed in nine MPU cycles. Everything else being equal, the second approach would probably be better.

## Programmed Review

27.	An MPU cycle is considered to be the _____ (minimum/maximum) time required to fetch a data byte from memory.
28.	(minimum) The minimum amount of time required to fetch and execute any instruction is _____ MPU cycles.
29.	(two) In the case of a variable operand, _____ (immediate/direct) addressing would be the most practical addressing mode to use.
30.	(direct) In the direct addressing mode, the second of two instruction bytes is the address of the _____ (opcode/operand).
31.	(operand) Direct addressing generally requires _____ (more/less) memory and _____ execution times than immediate (shorter/longer) addressing.
32.	(more/longer) Immediate and direct addressing modes _____ be combined in the same program. (may/may not)
	(may)

## EXPERIMENT 5

Perform Experiment 5 in Unit 12 of this course. After you finish the experiment, return to this unit and complete your studies.

## BINARY ARITHMETIC

Before you proceed with the more advanced aspects of microprocessors, you must understand how “computer arithmetic” is used to manipulate numbers. Therefore, the last section of this unit will address these basic principles.

A number system can be used to perform two basic operations: addition and subtraction. But by using addition and subtraction, you can then perform multiplication, division, and any other numerical operation. For simplicity, we will use decimal arithmetic as a guide.

### Binary Addition

Binary addition is performed somewhat like decimal addition. If two decimal numbers,  $56719_{10}$  and  $31863_{10}$  for example, are added together, the sum  $88582_{10}$  is obtained. You can analyze the details of this operation in the following manner.

NOTE: In the following explanations, the term “first column” refers to the first column of figures you work with in the problem — the column on the right (9, 3, and 2 in the following example). The term “second column” refers to the second column you work with, etc.

Carry:	00101
Addend:	56719
Augend:	+ 31863
Sum:	<u>88582</u>

Adding the first column, decimal numbers 9 and 3, gives the sum of 12. This is expressed in the sum as the digit 2 with a carry of 1. The carry is then added to the next column. Adding the second column decimal numbers 1 and 6, and the carry from the first column, 1, gives the sum of 8, with no carry. This process continues until all of the columns (including carries) have been added. The sum represents the numeric value of the addend and the augend. (The addend is the number to be added to another number, while the augend is the number to which the addend is added.)



When you add two binary numbers, you perform the same operation. The example below summarizes the four rules of addition with binary numbers.

1.  $0 + 0 = 0$
2.  $0 + 1 = 1$
3.  $1 + 1 = 0$  with a carry of 1
4.  $1 + 1 + 1 = 1$  with a carry of 1

To illustrate the process of binary addition, let's add  $1101_2$  to  $1101_2$ .

Carry:	1101
Addend:	1101 <sub>2</sub>
Augend:	+ 1101 <sub>2</sub>
Sum:	<u>11010<sub>2</sub></u>

In the first column, 1 plus 1 equals 0 with a carry of 1 to the second column. This agrees with rule 3.

In the second column, 0 plus 0 equals 0 with no carry. The carry from the first column is added to this. Thus, 0 plus 1 equals 1 with no carry. These two additions in the second column give a total sum of 1 with a carry of 0. Rules 1 and 2 were used to obtain the sum.

In column three, 1 plus 1 equals 0 with a carry of 1. To this sum, the second column is added. This yields a third column sum of 0 with a carry of 1 to column four. Rules 3 and 1 were used to obtain the sum.

In column four, 1 plus 1 equals 0 with a carry of 1. To this sum, the third column carry is added. This yields a fourth column sum of 1 with a carry to the fifth column. Rule 4 allows you to add three binary 1's and obtain 1 with a carry of 1.

In column five, there is no addend or augend. Therefore, you can assume rule 2 and add the carry to obtain the sum of 1. Thus, the sum of  $1101_2$  plus  $1101_2$  equals  $11010_2$ . You can verify this by converting the binary numbers to decimal numbers.

Now study the following two examples of binary addition, where  $10001111_2$  is added to  $10110101_2$  and  $111011_2$  is added to  $11001100_2$ .

$$\begin{array}{r}
 \text{Carry:} \quad 10111111 \\
 \text{Addend:} \quad 10110101_2 \\
 \text{Augend:} \quad + 10001111_2 \\
 \hline
 \text{Sum:} \quad 101000100_2
 \end{array}$$

$$\begin{array}{r}
 \text{Carry:} \quad 11111000 \\
 \text{Addend:} \quad 11001100_2 \\
 \text{Augend:} \quad + 00111011_2 \\
 \hline
 \text{Sum:} \quad 100000111_2
 \end{array}$$

When a microprocessor adds binary numbers, 8-bit numbers are generally used. As shown in the last example, two zeros were added after the MSB of the augend to produce an 8-bit number. After addition, a 1 in the ninth bit is represented as the “carry” bit by the microprocessor.

## Binary Subtraction

Binary subtraction is performed exactly like decimal subtraction. Therefore, before you attempt binary subtraction, you should reexamine decimal subtraction. You know that in decimal arithmetic, if 5486 is subtracted from 8303, the difference, 2817 is obtained.

$$\begin{array}{r}
 \text{Minuend after borrow} \quad 7 \quad 12 \quad 9 \quad 13 \\
 \text{Minuend:} \quad 8 \quad 3 \quad 0 \quad 3 \\
 \text{Subtrahend:} \quad - \quad 5 \quad 4 \quad 8 \quad 6 \\
 \hline
 \text{Difference:} \quad 2 \quad 8 \quad 1 \quad 7
 \end{array}$$

Because the digit 6 in the subtrahend is larger than the digit 3 in the minuend, a 1 is borrowed from the next high-order digit in the minuend. If that digit is a 0, as in this example, 1 is borrowed from the next high-order digit that contains a number other than 0. That digit is reduced by 1 (from 3 to 2 in our example) and the digits skipped in the minuend are given the value 9. This is equivalent to removing 1 from 30 with the result of 29, as in our example. In the decimal system, the digit borrowed has the value of 10. Therefore, the minuend digit now has the value 13, and 6 from 13 equals 7.

In the second column, 8 from 9 equals 1. Since the subtrahend is larger than the minuend in the third column, 1 is borrowed from the next higher-order digit. This raises the minuend value from 2 to 12, and 4 from 12 equals 8. In the fourth column, the minuend was reduced from 8 to 7 because of the previous borrow, and 5 from 7 equals 2.

Whenever 1 is borrowed from a higher-order digit, the borrow is equal in value to the radix or base of the number system. As you know, the radix or base of the decimal number system is 10, and the radix or base in the binary system is 2. Therefore, a borrow in the decimal number system equals 10, while a borrow in the binary number system equals 2.

When you subtract one binary number from another, you use the same method described for decimal subtraction. This is summarized by the following for binary subtraction.

1.  $0 - 0 = 0$
2.  $1 - 1 = 0$
3.  $1 - 0 = 1$
4.  $0 - 1 = 1$  with a borrow of 1.

To illustrate the process of binary subtraction, let's subtract  $1101_2$  from  $11011_2$ .

Minuend after borrow:	0 10 10 1 1
Minuend:	1 1 0 1 1
Subtrahend:	— 1 1 0 1
Difference:	<u>1 1 1 0</u>

The "minuend after borrow" now shows the value of each minuend digit after a borrow occurs. **Remember that binary 10 equals decimal 2.**

In the first column, 1 from 1 equals 0 (rule 2). Then, 0 from 1 in the second column equals 1 (rule 3). In the third column, 1 from 0 requires a borrow from the fourth column. Thus, 1 from 10 equals 1 (rule 4). The minuend in the fourth column is now 0, from the previous borrow. Therefore, a borrow is required from the fifth column, so that 1 from 10 in the fourth column equals 1 (rule 4). Because of the previous borrow, the minuend in the fifth column is now 0 and the subtrahend is 0 (nonexistent), so that 0 from 0 equals 0 (rule 1). The 0 in the fifth column is not shown in the difference because it is not a significant bit. Thus, the difference between  $11011_2$  and  $1101_2$  is  $1110_2$ . You can verify this by converting the binary number to a decimal number and subtracting.

As a further example of binary subtraction, subtract  $00100101_2$  from  $11000100_2$ , as shown below. Then proceed to the next example and subtract  $10111010_2$  from  $11101110_2$ .

Minuend after borrow:	1	0	1	1	1	10	1	10
Minuend:	1	1	0	0	0	1	0	0
Subtrahend:	–	0	0	1	0	0	1	0
Difference:		1	0	0	1	1	1	1

Minuend after borrow:	0	0	10	10	1	1	1	0
Minuend:	1	1	1	0	1	1	1	0
Subtrahend:	–	1	0	1	1	0	1	0
Difference:		0	0	1	1	0	1	0

When a borrow is required in the minuend, 1 is obtained from the next high-order bit that contains a 1. That bit then becomes 0, and all bits skipped (0 value bits) are given the value of 1. This is equivalent to removing 1 from  $1000_2$  with the result of  $0111_2$ .

As with binary addition, microprocessors generally subtract with 8-bit number groups. In the previous example, the answer contained only six significant bits, but two 0 bits were added to maintain the 8-bit grouping. This would also hold true for the minuend and subtrahend.

## Binary Multiplication

Multiplication is a short method of adding a number to itself as many times as it is specified by the multiplier. However, if you were to multiply  $324_{10}$  by  $233_{10}$  you would probably use the following method.

Multiplicand:	324
Multiplier:	× 223
First partial product:	<u>972</u>
Second partial product:	648
Third partial product:	648
Carry:	<u>0121</u>
Final product:	<u>72252</u>

Using the short form of multiplication, you multiply the multiplicand by each digit of the multiplier and then sum the partial products to obtain the final product. Note that, for convenience, the additive carries are set down under the partial products rather than over them as in normal addition.

Binary multiplication follows the same general principles as decimal multiplication. However, with only two possible multiplier bits (1 or 0), binary multiplication is a much simpler process. The example below lists the rules of binary multiplication. These rules will be used to multiply  $1111_2$  by  $1101_2$ .

1.  $0 \times 0 = 0$
2.  $0 \times 1 = 0$
3.  $1 \times 0 = 0$
4.  $1 \times 1 = 1$

Multiplicand:	1111
Multiplier:	$\times 1101$
First partial product:	1111
Second partial product:	0000
Carry:	0000
Sum of partial products:	1111
Third partial product:	1111
Carry:	111100
Sum of partial products:	1001011
Fourth partial product:	1111
Carry:	1111000
Final product:	11000011

As with decimal multiplication, you multiply the multiplicand by each bit in the multiplier and add the partial sums. First you multiply  $1111_2$  by the least significant multiplier bit (1) and set down the partial product so the least significant bit (LSB) is under the multiplier bit. Then you multiply the multiplicand by the next multiplier bit (0) and set down the partial product so the LSB is under the multiplier bit. Now that there are two partial products, they should be added. Although it is possible to add more than two binary numbers, keeping track of multiple carries may become confusing. Therefore, for these examples, add only two partial products at a time.

Notice that the first partial product is identical to the multiplicand. The second partial product is all zeros. Since the binary number system contains only ones and zeros, the partial product will always equal either the multiplicand or zero. Because of this, you can obtain the third partial product by copying the multiplicand. Begin with the LSB under the third multiplier bit. Add this value to the previous partial sum. Now obtain the fourth partial product by copying the multiplicand. Begin with the LSB under the fourth multiplier bit. Add this value to the previous partial sum. This is the final product. Again, you can verify the result by converting the binary numbers to decimal.



Reexamine the illustration for the previous multiplication example and you will notice that binary multiplication is a process of shift and add. For each 1 bit in the multiplier you copy down the multiplicand, beginning with the LSB under the bit. You can ignore any zeros in the multiplier. But do not make the mistake of setting down the multiplicand under the 0 bit.

To make sure you fully understand binary multiplication, multiply  $1001_2$  by  $1100_2$  and then multiply  $1101_2$  by  $1111_2$ .

Multiplicand:	1001
Multiplier:	$\times 1100$
First partial product:	<u>0000</u>
Second partial product:	<u>0000</u>
Carry:	<u>0000</u>
Sum of partial products:	00000
Third partial product:	<u>1001</u>
Carry:	<u>00000</u>
Sum of partial products:	100100
Fourth partial product:	<u>1001</u>
Carry:	<u>000000</u>
Final product:	1101100

Multiplicand:	1101
Multiplier:	$\times 1111$
First partial product:	<u>1101</u>
Second partial product:	<u>1101</u>
Carry:	<u>11000</u>
Sum of partial products:	100111
Third partial product:	<u>1101</u>
Carry:	<u>100100</u>
Sum of partial products:	1011011
Fourth partial product:	<u>1101</u>
Carry:	<u>1111000</u>
Final product:	11000011

In the first of these examples, the two zeros in the multiplier were included in the multiplication process. This was to insure that the multiplicand was copied down under the proper multiplier bits. The multiplication process could have been represented in this manner:

Multiplicand:	1001
Multiplier:	$\times 1100$
Third partial product:	<u>100100</u>
Fourth partial product:	<u>1001</u>
Carry:	<u>000000</u>
Final product:	1101100

Remember, just as in decimal multiplication, you must keep track of any zeros by setting a zero in the product under the 0 bit in the multiplier. This is very important when the zero occupies the LSB.

## Binary Division

Division is the reverse of multiplication. Therefore, it is a procedure for determining how many times one number can be subtracted from another. The process you are probably familiar with is called “long” division. If you were to divide decimal 181 by 45, you would obtain the quotient,  $4\text{-}1/45$ , as follows:

Divisor 45	$\begin{array}{r} 004 \\ \sqrt{181} \\ 180 \\ \hline 1 \end{array}$	Quotient Dividend  Remainder
------------	---	---------------------------------------

Using long division, you would examine the most significant digit in the dividend and determine if the divisor was smaller in value. In this example, the divisor is larger, so the quotient is zero. Next, you examine the two most significant digits, and here again, the divisor is larger, so the quotient is again zero. Finally, you examine the whole dividend and discover it is approximately four times the divisor value. Therefore, you give the quotient a value of 4. Next, you subtract the product of 45 and 4 (180) from the dividend. The difference of 1 represents a fraction of the divisor. This fraction is added to the quotient to produce the correct answer of  $4\text{-}1/45$ .

Binary division is performed in a similar manner. However, binary division is a simpler process since the number base is two rather than ten. First, let's divide  $100011_2$  by  $101_2$ .

Divisor: 101	$\begin{array}{r} 000111 \\ \sqrt{100011} \\ 101 \phantom{000} \\ \hline 111 \phantom{00} \\ 101 \phantom{00} \\ \hline 101 \phantom{00} \\ 101 \phantom{00} \\ \hline 0 \end{array}$	Quotient Dividend  Remainder  Remainder  Remainder
--------------	---	---

Using long division, you examine the dividend beginning with the MSB and determine the number of bits required to exceed the value of the divisor. When you find this value, place a 1 in the quotient and subtract the divisor from the selected dividend value. Then carry the next least significant bit in the dividend down to the remainder. If you can subtract

the divisor from the new remainder, place a 1 in the quotient. Then subtract the divisor from the remainder and carry the next least significant bit in the dividend (LSB in this example) down to the remainder. If the divisor can be subtracted from the new remainder, place a 1 in the quotient and subtract the divisor from the remainder. Continue the process until all of the dividend bits have been carried down. Then express any remainder as a fraction of the divisor in the quotient. Thus,  $100011_2$  divided by  $101_2$  equals  $111_2$ . You can verify the answer by converting the binary numbers to decimal.

To make sure you fully understand binary division, work out the following examples of long division. Divide  $101000_2$  by  $1000_2$  and then divide  $100111_2$  by  $110_2$ .

	000101	Quotient
Divisor 1000	$\begin{array}{r} \overline{)101000} \\ 1000 \downarrow \\ \hline 1000 \\ 1000 \\ \hline 0 \end{array}$	Dividend
		Remainder
		Remainder

	000110.1	Quotient
Divisor 110	$\begin{array}{r} \overline{)100111.0} \\ 110 \downarrow \\ \hline 111 \\ 110 \downarrow \\ \hline 110 \\ 110 \\ \hline 0 \end{array}$	Dividend
		Remainder
		Remainder
		Remainder

In the second example, the quotient was not a whole number, but rather a whole number plus a fraction (remainder divided by the divisor). The answer  $110-11/110$  is correct. You could have left the answer in this form or, as in the example, continue the division process until the remainder was zero. This is made possible by adding a sufficient number of zeros after the binary point to permit division by the divisor. In the previous example, only one zero was added after the binary point. As in the decimal number system, adding zeros after the binary point, in the binary number system, will not affect the value of the number. Note that some numbers cannot be solved in this manner (e.g., decimal  $1/3$ ).

## Representing Negative Numbers

Until now, we have been examining binary arithmetic using unsigned numbers. However, when you perform some arithmetic operations with a microprocessor, you must be able to express both positive and negative (signed) numbers. Over the years, three methods have been developed for representing signed numbers. Of these, only one method has survived. The two older methods will be briefly examined first, followed by the system that is used today.

### SIGN AND MAGNITUDE

Using this system, a binary number contained both the sign (+ or -) and the value of the number. Therefore, positive and negative values were expressed as follows:

$$\begin{array}{rcl} +45_{10} & = & \underline{0} \quad 0101101 \\ & \swarrow & \quad \searrow \\ & \text{Sign} & \quad \text{Magnitude} \\ & \quad \quad & \quad \quad \quad \text{(MSB)} \\ -45_{10} & = & \underline{1} \quad 0101101 \end{array}$$

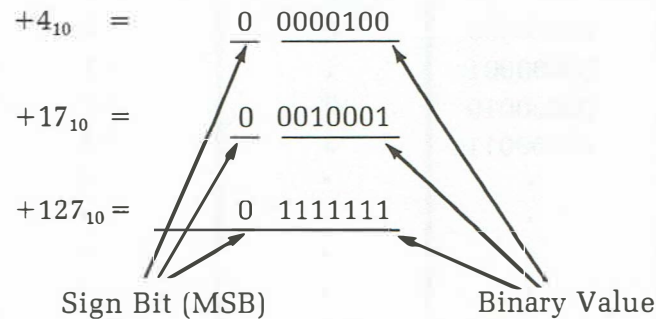
The MSB of the binary number indicated the sign, while the remaining bits contained the value of the number. As you can see, a zero sign bit indicated a positive value, while a one sign bit indicated a negative value.

While this method of representing negative numbers may seem logical, its popularity was short lived. Because it required complex and slow arithmetic circuitry, it was abandoned long before microprocessors were invented.



## ONE'S COMPLEMENT

Another method of representing negative numbers became popular in the early days of computers. It was called the one's complement method. Using this system, positive numbers were represented in the same way as in the sign-magnitude system. That is, the MSB in any number was considered to be a sign bit. A sign bit of 0 represented positive. Using 8-bit numbers, positive values were represented like this:



Negative numbers were represented by the one's complement of the positive value. The one's complement of a number is formed by changing all the 0's to 1's and all the 1's to 0's. As shown above,  $+4_{10}$  is represented as 0 0000100. By changing all 0's to 1's and all 1's to 0's, the representation for  $-4_{10}$  was formed. In this case:

$$-4_{10} = \underline{1} \underline{1111011}_2$$

Notice that all the bits, including the sign bit, were inverted. In the same way:

$$-17_{10} = \underline{1} \underline{1101110}_2$$

$$-127_{10} = \underline{1} \underline{0000000}_2$$

The one's complement method is not used for representing signed numbers in microprocessors, but if you need to find the one's complement of a number, simply change all the 0's to 1's and all the 1's to 0's.

Figure 4-40 shows an interesting relationship. In the first column, 8-bit patterns of 0's and 1's are shown. The second column shows the decimal number that each pattern represents if you consider the pattern to be an unsigned binary number. Notice that an 8-bit pattern can represent unsigned numbers between 0 and  $255_{10}$ .

BIT PATTERN	UNSIGNED BINARY	1's COMPLEMENT
00000000	0	+0
00000001	1	+1
00000010	2	+2
00000011	3	+3
.	.	.
.	.	.
.	.	.
.	.	.
01111100	124	+124
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	-127
10000001	129	-126
10000010	130	-125
10000011	131	-124
.	.	.
.	.	.
.	.	.
.	.	.
11111100	252	-3
11111101	253	-2
11111110	254	-1
11111111	255	-0

Figure 4-40  
Table of bit pattern values for unsigned binary numbers and  
1's complement numbers.

The third column shows the decimal number that each pattern represents if you consider the pattern to be a one's complement binary number. Notice that the range of numbers is from  $-127_{10}$  to  $+127_{10}$ . Notice also that there are two representations of zero. The pattern  $0000\ 0000_2$  represents +0 while its one's complement ( $1111\ 1111_2$ ) represents -0.

## TWO'S COMPLEMENT

The method used to represent signed numbers in microprocessors is called two's complement. In this system, positive numbers are represented just as they were with the sign-and-magnitude method and the one's complement method. That is, it uses the same bit pattern for all positive values up to  $+127_{10}$ . However, negative numbers are represented as the two's complement of positive numbers.

The two's complement of a number is formed by taking the one's complement and then adding 1. For example, if you work with 8-bit numbers and use the two's complement system,  $+4_{10}$  is represented by  $0000100_2$ . To find  $-4_{10}$ , you must take the two's complement of this number. You do this by first taking the one's complement, which is  $1111011_2$ . Next, add 1 to form the two's complement:

$$\begin{array}{r} 1111011_2 \\ + \quad 1 \\ \hline 1111100_2 \end{array}$$

Thus, the two's complement representation of  $-4_{10}$  is  $1111100_2$ .

To be sure you have the idea, look at a second example. How do you express  $-17_{10}$  as an 8-bit two's complement number? Start with binary representation of  $+17_{10}$ , which is  $00010001_2$ . Take the one's complement by changing all the 0's to 1's and 1's to 0's. Thus, the one's complement of  $+17_{10}$  is  $11101110_2$ . Next find the two's complement by adding 1:

$$\begin{array}{r} 11101110_2 \\ + \quad 1 \\ \hline 11101111_2 \end{array}$$

Figure 4-41 compares unsigned two's complement and one's complement numbers. Several 8-bit patterns are shown in the left column, while the other three columns show the decimal number represented by these patterns.

BIT PATTERN	UNSIGNED BINARY	2's COMPLEMENT	1's COMPLEMENT
00000000	0	0	+0
00000001	1	+1	+1
00000010	2	+2	+2
00000011	3	+3	+3
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
01111100	124	+124	+124
01111101	125	+125	+125
01111110	126	+126	+126
01111111	127	+127	+127
10000000	128	-128	-127
10000001	129	-127	-126
10000010	130	-126	-125
10000011	131	-125	-124
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
11111100	252	-4	-3
11111101	253	-3	-2
11111110	254	-2	-1
11111111	255	-1	-0

Figure 4-41  
Table of bit pattern values for unsigned binary,  
2's complement and 1's complement numbers.

Notice that the range of 8-bit two's complement numbers is from  $-128_{10}$  to  $+127_{10}$ . Notice also that there is only one representation for 0.

If this table included all  $256_{10}$  possible 8-bit patterns, you could look up any pattern to see what number it represents. The patterns that have 0 as their MSB are easy to determine without a table (the pattern represents the binary number directly). But what decimal number is represented by the two's complement number  $11110011_2$ ? You should know that this represents some negative number because the MSB is a 1.

Actually, you can determine the value very easily by taking the two's complement to find the equivalent positive number. Remember, you find the two's complement by taking the one's complement and adding 1. The one's complement is  $00001100_2$ . Thus, the two's complement is:

$$\begin{array}{r} 00001100_2 \\ + \quad \quad 1 \\ \hline 00001101_2 \quad \text{or } +13_{10} \end{array}$$

Since the two's complement of  $11110011_2$  represents  $+13_{10}$ , then  $11110011_2$  must equal  $-13_{10}$ .



## Programmed Review

Complete the following programmed review; then, check your answers with the correct answers on the succeeding page.

33. \_\_\_\_\_ and \_\_\_\_\_ are the two basic operations that can be performed with a number system.

34. Add the following binary numbers.

A.	$\begin{array}{r} 10011011 \\ +00010111 \\ \hline \end{array}$	B.	$\begin{array}{r} 11000110 \\ +00110001 \\ \hline \end{array}$	C.	$\begin{array}{r} 10000110 \\ +00110110 \\ \hline \end{array}$
----	--	----	--	----	--

35. Subtract the following binary numbers.

A.	$\begin{array}{r} 11011011 \\ -10110010 \\ \hline \end{array}$	B.	$\begin{array}{r} 10001011 \\ -10000001 \\ \hline \end{array}$	C.	$\begin{array}{r} 11011001 \\ -00111011 \\ \hline \end{array}$
----	--	----	--	----	--

36. Multiply the following binary numbers.

A.	$\begin{array}{r} 1011 \\ \times 1101 \\ \hline \end{array}$	B.	$\begin{array}{r} 1101 \\ \times 1001 \\ \hline \end{array}$	C.	$\begin{array}{r} 1100 \\ \times 1100 \\ \hline \end{array}$
----	--	----	--	----	--

37. Solve for the quotient in the following groups.

A.	$101 \overline{)1001011}$	B.	$11 \overline{)111001}$	C.	$1101 \overline{)11110111}$
----	---------------------------	----	-------------------------	----	-----------------------------

38.  $10001111_2$  represents decimal \_\_\_\_\_ in sign/magnitude notation.

39. The one's complement of  $00010110_2$  is \_\_\_\_\_.

40. The two's complement of  $00010110_2$  is \_\_\_\_\_.

41. The two's complement number  $11100110$  represents the decimal number \_\_\_\_\_.

42. Find the signed decimal equivalents of the following two's complement numbers.

<u>Two's Complement Number</u>	<u>Decimal Number</u>
00000111	
10000111	
11111111	
01110000	
10000000	

43. Find the two's complement representation for the following signed decimal numbers.

<u>Decimal Number</u>	<u>Two's Complement Number</u>
+32	
-32	
+73	
- 7	
-120	

## Answers To Programmed Review

33. Addition, subtraction.

34. A.

Carry:		00011111
Addend:		10011011
Augend:	+	00010111
Sum:		<u>10110010</u>

B. 11110111.

C, 10111100.

35. A. Minuend after borrow: 1 0 10 1 1 0 1 1  
 Minuend: 1 1 0 1 1 0 1 1  
 Subtrahend: − 1 0 1 1 0 0 1 0  
 Difference: 1 0 1 0 0 1

B. 1010.

C. 10011110.

36. A. Multiplicand: 1011  
Multiplier:  $\times 1101$   
First partial product: 1011  
Second partial product: 00000  
Carry: 0000  
Sum of partial products: 01011  
Third partial product: 101100  
Carry: 01000  
Sum of partial products: 110111  
Fourth partial product: 1011000  
Carry: 1110000  
Final product: 10001111

B. 1110101.

C. 10010000

37. A. Divisor: 101

0001111	Quotient
1001011	Dividend
101	
1000	Remainder
101	
111	Remainder
101	
101	Remainder
101	
0	Remainder

B. 10011.

C. 10011.

38. -15.

39. 11101001<sub>2</sub>.

40. 11101010<sub>2</sub>.

41. First, find the two's complement of 11100110 by changing 1's to 0's; 0's to 1's; and adding 1:

00011001
1
00011010

Since this number represents +26<sub>10</sub>, the original number must have represented -26<sub>10</sub>.

42. Two's Complement Number      Decimal Number

00000111	+7
10000111	-121
11111111	-1
01110000	+112
10000000	-128

43. Decimal Number      Two's Complement Number

+32	00100000
-32	11100000
+73	01001001
-7	11111001
-120	10001000

## TWO'S COMPLEMENT ARITHMETIC

In the previous discussion you saw that signed numbers are represented in microprocessors in two's complement form. Now you will see why.

In digital electronic devices, such as computers, simple circuits cost less and operate faster than more complex ones. Two's complement numbers are used with arithmetic because they allow the simplest, cheapest, and fastest circuits.

A characteristic of the two's complement system is that both signed and unsigned numbers can be added by the same circuit. For example, suppose you wish to add the unsigned numbers  $132_{10}$  and  $14_{10}$ . The addition would look like this:

$$\begin{array}{r}
 \text{Addend:} \quad 10000100 \quad 132_{10} \\
 \text{Augend:} \quad 00001110 \quad + \quad 14_{10} \\
 \hline
 \text{Sum:} \quad 10010010 \quad 146_{10}
 \end{array}$$

As you know, the microprocessor has an ALU circuit that can add unsigned binary numbers in this manner. The adder of the ALU is designed so that when the bit pattern  $10000100_2$  appears at one input and  $00001110_2$  appears at the other, the bit pattern  $10010010_2$  appears at the output.

The question arises, "How does the ALU know that the bit patterns at the inputs represent unsigned numbers and not two's complement numbers?" The answer is simple, "it doesn't!"

The ALU always adds as if the inputs were unsigned binary numbers. Nevertheless, it still produces the correct sum even if the inputs are signed two's complement numbers.

Look at the example given above. If you assume that the inputs are two's complement signed numbers, then the addend, augend, and sum are:

$$\begin{array}{r}
 \text{Addend:} \quad 10000100_2 \quad -124_{10} \\
 \text{Augend:} \quad 00001110_2 \quad + \quad 14_{10} \\
 \hline
 \text{Sum:} \quad 10010010_2 \quad -110_{10}
 \end{array}$$



Notice that the bit patterns are the same. Only the meaning of the bit patterns have changed. In the first example, we assumed that the bit patterns represented unsigned numbers and the adder produced the proper unsigned result. In the second example, we assumed that the bit patterns represented signed numbers. Again, the adder produced the proper signed result.

This proves a very important point. The adder in the ALU always adds bit patterns as if they are unsigned numbers. It is our interpretation of these bit patterns that decides if unsigned or signed numbers are indicated. The advantage of two's complement is that the bit patterns can be interpreted either way. This allows us to work with either signed or unsigned numbers without requiring different circuits for each.

Two's complement arithmetic also simplifies the arithmetic logic unit in another way. All microprocessors have a subtract instruction. Thus, the ALU must be able to subtract one number from another. However, if this required a separate subtraction circuit, the ALU would be more complex and costly. Fortunately, two's complement arithmetic allows the ALU to subtract using an adder circuit. That is, the MPU uses the same circuit to add and subtract.

The MPU subtracts by a binary addition process. To see how this works, it may be helpful to look at a similar process with the decimal number system. The decimal equivalent of two's complement is called ten's complement. Since you are more familiar with the decimal number system, let's briefly examine ten's complement arithmetic.

## Ten's Complement Arithmetic

An easy way to illustrate ten's complement is to consider an analogy. Visualize an automobile odometer or mileage indicator. Generally, this is a 6-digit device that indicates mileage between 00,000.0 and 99,999.9 miles. Let's ignore the tenths digit and concentrate on the other five.

In an automobile, the odometer generally operates in only one direction, forward. However, consider what happens if it is turned backwards instead. Starting at +3 miles, the count would proceed backwards as follows:

00,003  
00,002  
00,001  
00,000  
99,999  
99,998  
99,997  
etc.

It is easy to visualize that 99,999 represents  $-1$  mile. Also 99,998 represents  $-2$  miles; 99,997 represents  $-3$  miles; etc. This is how signed numbers are represented in ten's complement form.

Once you accept this system for representing positive and negative numbers, you can perform arithmetic with these signed numbers. For example, if you add  $+3$  and  $-2$ , the result is  $+1$ . Using the system developed above,  $+3$  is represented by 00003 while  $-2$  is represented by 99,998. Thus, the addition looks like this:

$$\begin{array}{r}
 00003 \quad +3 \\
 + 99998 \quad -2 \\
 \hline
 100001 \quad +1
 \end{array}$$

Discard final carry  $\xrightarrow{\quad}$

If you now discard the final carry on the left side of the sum, the answer is 00001, the representation of  $+1$ . You can also find the ten's complement of a digit by subtracting the digit from ten. For example, the ten's complement of 6 is 4 since  $10 - 6 = 4$ . To complement a number containing more than one digit, raise ten to a power equal to the total number of digits, then subtract the number from it. As an example, to obtain the ten's

complement of  $654_{10}$  first raise ten to the third power since there are three digits in the number. Then subtract 654 from the result.

$$\begin{array}{r} 10^3 = 1000 \\ - 654 \\ \hline 346 \end{array}$$

Thus, the ten's complement of  $654_{10}$  is  $346_{10}$ .

Once you find the ten's complement, you can subtract one number from another by an indirect method using only addition. Most of us have learned to subtract like this:

$$\begin{array}{r} \text{Minuend:} \quad 973 \\ \text{Subtrahend:} \quad -654 \\ \hline \text{Difference:} \quad 319 \end{array}$$

However, you can arrive at the same answer by using ten's complement of the subtrahend and adding. Recall that the ten's complement of  $654_{10}$  is  $346_{10}$ . Let's compare these two methods of subtraction:

#### STANDARD METHOD

$$\begin{array}{r} \text{Minuend} \quad 973 \\ \text{Subtrahend} \quad -654 \\ \hline \text{Difference} \quad 319 \end{array}$$

#### TEN'S COMPLEMENT METHOD

$$\begin{array}{r} \text{Minuend} \quad 973 \\ + \text{Ten's complement of Subtrahend} \quad +346 \\ \hline \text{Difference} \quad 1319 \end{array}$$

↑ Discard final carry

Notice that when you use the ten's complement method, the answer is too large, by  $1000_{10}$ . However, you can still arrive at the correct answer by simply discarding the final carry.

While the ten's complement method of subtraction works, it is not readily used because it is more complex than the standard method. In fact, it does not eliminate subtraction entirely since the ten's complement itself is found by subtraction.

The binary equivalent of ten's complement is two's complement. It overcomes the disadvantage of ten's complement in that the two's complement can be formed without any subtraction at all. Recall that you can form the two's complement of a binary number by changing all the 0's to 1's and all the 1's to 0's and then adding. Let's examine two's complement arithmetic in more detail.

## Two's Complement Subtraction


As in ten's complement arithmetic, you can form the two's complement by subtracting from a power of the base or radix (two). However, because the MPU cannot subtract directly, it uses the method given earlier for finding the two's complement. Once the two's complement is formed, the MPU can subtract indirectly by adding the two's complement of the subtrahend to the minuend.

To illustrate this point, observe the following two ways of subtracting  $26_{10}$  from  $69_{10}$ . The two numbers are expressed as they would appear to an 8-bit microprocessor. The standard method of subtraction looks like this:

Minuend	$01000101_2$	69
Subtrahend	$-00011010_2$	-26
Difference	$00101011_2$	43

While this method works fine on paper, it's of little use to the microprocessor since the MPU has no subtraction circuitry. However, the MPU can still perform subtraction by the indirect method of adding the two's complement of the subtrahend to the minuend:

	Minuend	$01000101$
Two's complement of	Subtrahend	$+11100110$
	Difference	$100101011$

Discard final carry 

This illustrates a major reason for using the two's complement system to represent signed numbers. It allows the MPU to subtract and add with the same circuit.

How microprocessors subtract is of little importance to the people who use them. Most microprocessors have a subtract instruction. This instruction is used like any other, without regard for how the operation is implemented internally. When the subtract instruction is implemented, the MPU automatically takes care of operations like complementing the subtrahend, adding, and discarding the carry. The procedure has been explained here so you can appreciate the importance of two's complement arithmetic.

## Arithmetic With Signed Numbers

There are many applications in which the microprocessor must work with signed numbers. In these cases, signed numbers are represented in two's complement form. While this greatly simplifies the circuitry of the MPU, it places an extra burden on the user. The programmer must ensure that all signed numbers are entered into the microprocessor in two's complement form. Also, the resulting data produced by the MPU may be in two's complement form. Here's how an 8-bit MPU handles signed numbers.

### ADDING POSITIVE NUMBERS

Assume that the MPU is to add the two positive numbers +7 and +3. Since an 8-bit MPU is assumed, the arithmetic operation looks like this:

$$\begin{array}{r}
 \underline{00000111} \qquad +7 \\
 + \underline{00000011} \qquad +3 \\
 \hline
 \underline{00001010} \qquad +10
 \end{array}$$

The sign bits are underlined. Remember, with signed numbers, the MSB is the sign bit. A 0 represents "+" and a 1 represents a "-." In this example, you added +7 and +3 to form a sum of +10<sub>10</sub>. You know that all three numbers are positive since the MSB's are all 0's.

While the operation seems straightforward enough, it is easy to make an error when adding positive numbers. Remember, the highest 8-bit positive number you can represent in two's complement form is +127<sub>10</sub>. If the sum exceeds this value, an error occurs. For example, suppose you attempt to add +65<sub>10</sub> to +67<sub>10</sub>. The MPU adds the numbers as if they were unsigned binary:

$$\begin{array}{r}
 \underline{01000001} \\
 + \underline{01000011} \\
 \hline
 \underline{10000100}
 \end{array}$$


If the answer is interpreted as a two's complement number, an error has occurred. You have added two positive numbers and yet the answer appears to be negative since the MSB of the sum is 1. This is called two's complement overflow. It occurs when the sum exceeds +127<sub>10</sub>. Many microprocessors have a way of detecting this condition, which we will discuss later.



### ADDING POSITIVE AND NEGATIVE NUMBERS

The real advantage of the two's complement system is best illustrated when you add numbers with unlike signs. For example, assume that an 8-bit microprocessor is to add +7 and -3. Remember, since these are signed numbers, they must be represented in two's complement form. That is, +7 is represented as  $00000111_2$  while -3 is represented as  $11111101_2$ . If these two numbers are added, the sum will be:

Addend	00000111	(+7)
Augend	+ 11111101	+ (-3)
Sum	<u>100000100</u>	(+4)

 Discard final carry

Notice that the sum is correct if you ignore the final carry bit. Keep in mind that the MPU adds the two numbers as if they were unsigned binary numbers. It is merely our interpretation of the answer that makes the system work for signed numbers.

The system also works when the negative number is larger. For example, when -9 is added to +8 the result should be -1. Remember, the signed numbers must be represented in two's complement form:

Addend	11110111	(-9)
Augend	+ 00001000	+ (+8)
Sum	<u>11111111</u>	(-1)


Notice that the sum is the two's complement representation for -1.

## ADDING NEGATIVE NUMBERS

The final case involves two negative numbers. If both numbers are negative, then the sum should also be negative.

For example, suppose the MPU is to add  $-3$  to  $-4$ . Obviously, the result should be  $-7$ . The two signed numbers must be represented in two's complement form. That is,  $-3$  must be represented as  $11111101_2$  while  $-4$  must be represented as  $11111100_2$ . The MPU adds these two bit patterns as if they were unsigned binary numbers. Thus the result is:

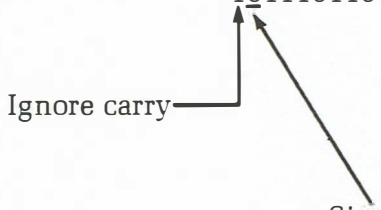
Addend	11111101	(-3)
Augend	+ 11111100	+ (-4)
Sum	<u>111111001</u>	<u>(-7)</u>


 Discard final carry

Once again, the answer is correct if you ignore the final carry bit.

When you add two negative numbers, you must remember the capacity of the MPU. The largest negative number that can be represented by 8 bits is  $-128_{10}$ . If the sum exceeds this value, the sum will appear to be in error. For example, suppose you add  $-120_{10}$  to  $-18_{10}$ .

	10001000	(-120)
	+ 11101110	+ (-18)
	<u>101110110</u>	<u>(-138)</u>


 Ignore carry

Sign bit

Notice that the sign bit in the sum is 0, representing a positive number. Thus, the MPU has added two negative numbers and has produced a positive result. This apparent error is caused by exceeding the 8-bit capacity of the microprocessor. This is another example of two's complement overflow.

## Programmed Review

Complete the following programmed review; then, check your answers with the correct answers that follow.

- |     |   |
|-----|---|
| 44. | In microprocessors, signed numbers are represented in _____ form.   |
| 45. | The ALU adds bit patterns as if they represent _____ binary numbers.  |
| 46. | When a microprocessor executes a subtract instruction, what operations are actually performed inside the MPU? |
| 47. | What is the largest 8-bit positive number that can be represented in two's complement form?                   |
| 48. | When you are adding two positive numbers, what is meant by two's complement overflow?                         |

49. If  $+19_{10}$  and  $-21_{10}$  are added by an 8-bit microprocessor, the two's complement result will be \_\_\_\_\_.
50. Can two's complement overflow occur when two negative numbers are added?
51. A microprocessor adds  $10001110_2$  to  $00010001_2$ . If these are unsigned binary numbers, the resulting bit pattern will be \_\_\_\_\_. If these are two's complement numbers, the resulting bit pattern will be \_\_\_\_\_.
52. If the bit patterns in question 51 represent unsigned numbers, the resulting bit pattern represents decimal \_\_\_\_\_.
53. If the bit patterns in question 51 represent two's complement numbers, the resulting bit pattern represent decimal \_\_\_\_\_.

## Answers To Programmed Review

44. Two's complement.
45. Unsigned.
46. The following operations occur:
  1. The MPU complements the subtrahend by changing 0's to 1's and 1's to 0's.
  2. One is added to the complemented subtrahend to form the two's complement.
  3. The two's complement of the subtrahend is added to the minuend.
47.  $01111111_2$  or  $+127_{10}$ .
48. When you add positive numbers, two's complement overflow occurs when the sum exceeds  $+127_{10}$ .
49.  $11111110_2$  or  $-2_{10}$ .
50. Yes. When you add negative numbers, two's complement overflow occurs when the sum exceeds  $-128_{10}$ .
51. In either case, the resulting bit pattern will be 10011111.
52.  $159_{10}$ .
53.  $-97_{10}$ .



## BOOLEAN OPERATIONS

Along with the basic mathematical processes examined earlier, the microprocessor can manipulate binary numbers logically. This system was conceived using the theorems developed by mathematician George Boole. As a result, this branch of binary mathematics is given the name Boolean Algebra. In this section, the Boolean operations performed by the microprocessor will be examined. A more detailed description of Boolean Algebra is provided in the Heathkit/Zenith Education Systems course titled "Digital Techniques."

### AND Operation

The AND function produces the logical product of two or more logic variables. That is, the logic product of an AND operation is logic 1 if all of the variable inputs are logic 1. If any of the variable inputs are logic 0, the logical product is 0. This process can be represented by the formula  $(A \bullet B = C)$  where A and B represent input variables (logic 1 or 0) and C represents the output or logical product of the AND operation. The AND function is designated by a dot between variables. Do not confuse it with the mathematical multiplication sign.

Figure 4-42 is a "truth table" for a 2-variable AND function. The 1's and 0's represent all of the possible logic combinations. Thus, you can see that the AND function is a sort of "all-or-nothing" operation. Unless all the input variables are logic 1, the output logic cannot be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 4-42  
Truth table for a two-variable AND function.

When the microprocessor implements the logic AND operation, one 8-bit binary number is ANDed with a second 8-bit binary number. Refer to the example below for an illustration of this process.

	<u>8-BIT</u>		<u>8-BIT</u>		<u>RESULTS OF</u>	
	<u>NUMBER</u>		<u>NUMBER</u>		<u>AND OPERATION</u>	
MSB	1	•	1	=	1	MSB
	0	•	0	=	0	
	0	•	1	=	0	
	1	•	0	=	0	
	1	•	1	=	1	
	0	•	1	=	0	
	1	•	0	=	0	
LSB	0	•	0	=	0	LSB

Although more than two logic variables can be ANDed together, the microprocessor operates on only two variables at a time. Now try one more example of the AND operation. AND 10011101 with 11000110.

$$\begin{array}{l}
 1 \cdot 1 = 1 \quad \text{MSB} \\
 0 \cdot 1 = 0 \\
 0 \cdot 0 = 0 \\
 1 \cdot 0 = 0 \\
 1 \cdot 0 = 0 \\
 1 \cdot 1 = 1 \\
 0 \cdot 1 = 0 \\
 1 \cdot 0 = 0 \quad \text{LSB}
 \end{array}$$



## OR Operation

The OR function (more precisely, inclusive OR) produces the logical sum of two or more logic variables. That is, the logical sum of an OR operation is logic 1 if either input is logic 1. The logic sum is 0 if ALL of the input variables are logic 0. This process can be represented by the formula ( $A + B = C$ ) where A and B represent input variables and C represents the output or logical sum of the OR operation. The OR function is designated by a plus sign, or in some cases, a circle dot  $\odot$ , between the variables. Do not confuse the plus sign with the mathematical add sign.

Figure 4-43 is a “truth table” for a 2-variable OR function. The 1’s and 0’s represent all of the possible logic combinations. Thus, you can see that the OR function is a sort of “either or both” operation. If either or both input variables are logic 1, the output must be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4-43  
Truth table for a two-variable OR function.

When the microprocessor implements the logic OR operation, one 8-bit binary number is ORed with a second 8-bit binary number. This process can be seen in the example given below.

	8-BIT NUMBER		8-BIT NUMBER		RESULTS OF OR OPERATION	
MSB	1	+	1	=	1	MSB
	0	+	0	=	0	
	0	+	1	=	1	
	1	+	0	=	1	
	1	+	1	=	1	
	0	+	1	=	1	
	1	+	0	=	1	
LSB	0	+	0	=	0	

As with the AND function, two or more logic variables can be ORed together. However, the microprocessor operates only on two variables at a time. Now try one more example of the OR operation. OR  $10011101_2$  with  $11000101_2$ .

$$\begin{array}{l} 1 + 1 = 1 \text{ MSB} \\ 0 + 1 = 1 \\ 0 + 0 = 0 \\ 1 + 0 = 1 \\ 1 + 0 = 1 \\ 1 + 1 = 1 \\ 0 + 0 = 0 \\ 1 + 1 = 1 \text{ LSB} \end{array}$$

## Exclusive OR Operation

The Exclusive OR (EOR or XOR) function performs a logical test for “equality” between two logic variables. That is, if two variable inputs are equal (both logic 1 or 0), the output or result of the EOR operation is logic 0. If the inputs are not equal (one is logic 1, the other logic 0), the output is logic 1. This can be represented by the formula  $(A \oplus B = C)$  where A and B represent input variables and C represents the output or result. The EOR function is designated by a circled plus sign between the variables.

Figure 4-44 is a “truth table” for the EOR function. The 1’s and 0’s represent all of the possible logic combinations. You can see that the EOR function is a sort of “either but not both” operation. Hence, either input can be logic 1 or logic 0, but not both, for a logic 1 output.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4-44  
Truth table for a two-variable EOR function.

When the microprocessor implements the logic EOR operation, one 8-bit binary number is exclusively ORed with a second 8-bit binary number. This process is shown in the following example.

	8-BIT NUMBER		8-BIT NUMBER		RESULTS OF EOR OPERATION	
MSB	1	$\oplus$	1	=	0	MSB
	0	$\oplus$	0	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
	1	$\oplus$	1	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
LSB	0	$\oplus$	0	=	0	LSB

Now try one more example of the EOR operation. EOR  $10011101_2$  with  $11000101_2$ .

$$\begin{aligned}
 1 \oplus 1 &= 0 \text{ MSB} \\
 0 \oplus 1 &= 1 \\
 0 \oplus 0 &= 0 \\
 1 \oplus 0 &= 1 \\
 1 \oplus 0 &= 1 \\
 1 \oplus 1 &= 0 \\
 0 \oplus 0 &= 0 \\
 1 \oplus 1 &= 0 \text{ LSB}
 \end{aligned}$$

## Invert Operation

The invert operation performs a direct complement of a single input variable. That is, a logic 1 input will produce a logic 0 output and a logic 0 input will produce a logic 1 output. This process is represented by the "truth table" shown in Figure 4-45.

INPUT	OUTPUT
A	$\bar{A}$
1	0
0	1

Figure 4-45  
Truth table for an INVERT function.

Note that the complement of A is  $\bar{A}$ . The bar above the A indicates that A has been inverted, and is read "not A." Conversely, the complement of  $\bar{A}$  is A.

When the microprocessor implements the logic invert operation, the 8-bit binary number is complemented. This operation is also known as 1's complement. Thus, the complement of  $11010110_2$  is  $00101001_2$ . As with the previous logic operations, the invert function operates on each individual bit of the 8-bit number.



## Programmed Review

Complete the following programmed review; then, check your answers with the correct answers that follow.

**54.** The result of an AND operation is binary 1 when:

- A. All inputs are binary 0.
- B. Any one input is binary 0.
- C. All inputs are binary 1.
- D. Any one input is binary 1.

**55.** Perform the AND operation on the following 8-bit number pairs.

- A. 11010110 and 10000111.
- B. 00110011 and 11110000.
- C. 10101010 and 11011011.

**56.** The result of an OR operation is binary 0 when:

- A. All inputs are binary 1.
- B. All inputs are binary 0.
- C. Any one input is binary 1.
- D. Any one input is binary 0.

**57.** Perform the OR operation on the following 8-bit number pairs.

- A. 11010110 and 10000111.
- B. 00110011 and 11110000.
- C. 10101010 and 11011011.

**58.** The result of an XOR operation is binary 0 if the inputs are:

- A. Equal.
- B. Not equal.

59. The symbol for the EOR operation is:

- A.  $\cdot$
- B.  $+$
- C.  $\oplus$
- D.  $\times$

60. Perform the EOR operation on the following 8-bit number pairs.

- A. 11010110 and 10000111.
- B. 00110011 and 11110000.
- C. 10101010 and 11011011.

61.  $\bar{A}$  represents the \_\_\_\_\_ of A.

- A. Sum.
- B. Product.
- C. Complement.
- D. Supplement.

62. Perform the invert operation on the following 8-bit numbers.

- A. 11010110.
- B. 00110011.
- C. 10101010.

## Answers

54. C. All inputs are binary 1.

55. A.  $1 \cdot 1 = 1$   
 $1 \cdot 0 = 0$   
 $0 \cdot 0 = 0$   
 $1 \cdot 0 = 0$   
 $0 \cdot 0 = 0$   
 $1 \cdot 1 = 1$   
 $1 \cdot 1 = 1$   
 $0 \cdot 1 = 0$

B. 00110000.

C. 10001010.

56. B. All inputs are binary 0.

57. A.  $1 + 1 = 1$   
 $1 + 0 = 1$   
 $0 + 0 = 0$   
 $1 + 0 = 1$   
 $0 + 0 = 0$   
 $1 + 1 = 1$   
 $1 + 1 = 1$   
 $0 + 1 = 1$

B. 11110011.

C. 11111011.

58. A. Equal.

59. C.  $\oplus$

60. A.  $1 \oplus 1 = 0$   
 $1 \oplus 0 = 1$   
 $0 \oplus 0 = 0$   
 $1 \oplus 0 = 1$   
 $0 \oplus 0 = 0$   
 $1 \oplus 1 = 0$   
 $1 \oplus 1 = 0$   
 $0 \oplus 1 = 1$

B. 11000011.

C. 01110001.

61. C. Complement.

62. A. 00101001.

B. 11001100.

C. 01010101.

## EXPERIMENT 6

Perform Experiment 6 in Unit 12 of this course. After you finish the experiment, return to this unit and complete the Unit Examination.



## UNIT EXAMINATION

The following examination is designed to test your understanding of the material presented in this unit. Questions 1 — 15 are multiple choice; therefore, read each question and all four answers before you select the answer you feel is most correct. Questions 16 — 25 pertain to the arithmetic section of this unit, so you may need extra paper to complete this portion of the exam. When you have completed the examination, compare your answers with the correct ones that appear after the exam.

1. In microprocessor terminology, the number or piece of data that is operated upon is called the:
  - A. Operand.
  - B. Opcode.
  - C. Address.
  - D. Instruction.
  
2. The part of the instruction that tells the microprocessor what operation to perform is called the:
  - A. Operand.
  - B. Opcode.
  - C. Address.
  - D. Mnemonic.
  
3. The portion of the microcomputer in which instructions and data are stored is called the:
  - A. ALU.
  - B. MPU.
  - C. RAM.
  - D. Data bus.
  
4. An 8-bit byte in memory can represent an:
  - A. Operand.
  - B. Opcode.
  - C. Address.
  - D. All of the above.

5. During the fetch phase:
- A. The opcode is fetched from memory and is decoded.
  - B. The address of the operand is fetched from memory and decoded.
  - C. The operand is fetched from memory and operated upon.
  - D. The program count is fetched from memory.
6. In what register is the result of an arithmetic operation normally placed?
- A. The data register.
  - B. The address register.
  - C. The arithmetic logic unit (ALU).
  - D. The accumulator.
7. During the fetch and execute phases of the "load accumulator direct" instruction, the information on the data bus will be:
- A. The operand address followed by the operand.
  - B. The program count, followed by the opcode, followed by the operand address, followed by the operand.
  - C. The opcode, followed by the operand address, followed by the operand.
  - D. The opcode, followed by the operand.
8. In the immediate addressing mode, the second byte of the instruction is the:
- A. Opcode of the instruction.
  - B. Number that is to be operated upon.
  - C. Address of the operand.
  - D. Address of the opcode.
9. In the direct addressing mode, the second byte of the instruction is the:
- A. Opcode of the instruction.
  - B. Number that is to be operated upon.
  - C. Address of the operand.
  - D. Address of the opcode.

10. Which of the following is normally a 1-byte instruction?
- A. Halt.
  - B. Add immediate.
  - C. Load accumulator direct.
  - D. Store accumulator direct.
11. At the start of the fetch phase, the program counter contains:
- A. The address of the operand to be fetched.
  - B. The address of the next opcode to be fetched.
  - C. The opcode of the instruction.
  - D. The operand.
12. Which unit of the microprocessor holds the opcode while it is being decoded?
- A. The address register.
  - B. The accumulator.
  - C. The data register.
  - D. The program counter.
13. The program shown in Figure 4-46:

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	86	LDA
01	00	00
02	97	STA
03	09	09
04	97	STA
05	0A	0A
06	97	STA
07	0B	0B
08	3E	HLT
09	—	—
0A	—	—
0B	—	—

Figure 4-46  
Program for question 13.

- A. Adds the contents of memory location 09, 0A, and 0B.
- B. Stores 00 in locations 09, 0A, and 0B.
- C. Stores 09 in location 03, 0A in location 05, and 0B in location 07.
- D. Stores 0B in the accumulator.

14. The program shown in Figure 4-47:

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	96	LDA
01	09	09
02	9B	ADD
03	09	09
04	9B	ADD
05	09	09
06	9B	ADD
07	09	09
08	3E	HLT
09	04	04

Figure 4-47  
Program for question 14.

- Multiplies 4 times 4 and holds the product in the accumulator.
- Multiplies 9 times 3 and holds the product in the accumulator.
- Multiplies 4 times 3 and stores the product in the accumulator.
- Multiplies 9 times 4 and holds the product in the accumulator.

15. The program shown in Figure 4-48:

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS
00	96	LDA
01	0D	0D
02	97	STA
03	0F	0F
04	96	LDA
05	0E	0E
06	97	STA
07	0D	0D
08	96	LDA
09	0F	0F
0A	97	STA
0B	0E	0E
0C	E3	HLT
0D	AA	AA
0E	BB	BB
0F	—	—

Figure 4-48  
Program for Question 15.

- Swaps the contents of memory location 0D and 0E.
- Stores  $AA_{16}$  in locations 0D, 0E, and 0F.
- Stores  $BB_{16}$  in locations 0D, 0E, and 0F.
- Adds  $AA_{16}$  and  $BB_{16}$ , storing the sum at location 0F.

16. Using binary arithmetic, ADD  $1101_2$  to  $10010110_2$ .
17. Using binary arithmetic, SUBTRACT  $1011_2$  from  $10110110_2$ .
18. Using binary arithmetic, MULTIPLY  $1001_2$  by  $1100_2$ .
19. Using binary arithmetic, DIVIDE  $100111_2$  by  $110_2$ .
20. Using 2's complement arithmetic, ADD  $+75_{10}$  to  $-6_{10}$ .
21. Using 2's complement arithmetic, SUBTRACT  $-15_{10}$  from  $-85_{10}$ .
22. Logically AND  $11011010$  with  $10010110$ .
23. Logically OR  $11011010$  with  $10010110$ .
24. Logically EOR  $11011010$  with  $10010110$ .
25. Logically **INVERT**  $11011010$ .





## EXAMINATION ANSWERS

The page numbers after each answer show where you can find the discussion for questions 1 — 15. Also, answers 16 — 25 show the procedure for obtaining the correct answer.

1. A — Operand. [4-16]
2. B — Opcode. [4-23]
3. C — RAM. [4-20]
4. D — All of the above. [4-20]
5. A — The opcode is fetched from memory and decoded. [4-21]
6. D — The accumulator. [4-17]
7. C — The opcode, followed by the operand address,  
followed by the operand. [4-47]
8. B — Number that is to be operated upon. [4-42]
9. C — Address of the operand. [4-45]
10. A — Halt. [4-40]
11. B — The address of the next opcode to be fetched. [4-43]
12. C — The data register. [4-17]
13. B — Stores 00 in locations 09, 0A, and 0B. [4-27-]  
[4-63]
14. A — Multiplies 4 times 4 and holds the product  
in the accumulator. [4-27-]  
[4-63]
15. A — Swaps the contents of memory location 0D and 0E. [4-27-]  
[4-63]

- |                 |               |
|-----------------|---------------|
| 11111010        | Carry         |
| 01001011        | Addend        |
| + 11111010      | Augend        |
| <hr/> 101000101 | Sum           |
| 01000101        | Corrected sum |

$$01000101_2 = +69_{10}.$$

$$21. \quad +85_{10} = 01010101_2$$

$$-85_{10} = 10101011_2$$

$$+15_{10} = 00001111_2$$

$$-15_{10} = 11110001_2$$

$\begin{array}{r} 10101011 \\ -11110001 \\ \hline 10111010 \end{array}$	}	$\begin{array}{r} 00001111 \\ 10101011 \\ +00001111 \\ \hline 010111010 \\ 10111010 \end{array}$	Carry Addend Augend Sum Corrected sum
---	---	--	---

$$10111010_2 = -70_{10}.$$

$$22. \quad 1 \cdot 1 = 1$$

$$1 \cdot 0 = 0$$

$$0 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

$$1 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 1 = 1$$

$$0 \cdot 0 = 0$$

$$23. \quad 1 + 1 = 1$$

$$1 + 0 = 1$$

$$0 + 0 = 0$$

$$1 + 1 = 1$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$1 + 1 = 1$$

$$0 + 0 = 0$$

$$24. \quad 1 \oplus 1 = 0$$

$$1 \oplus 0 = 1$$

$$0 \oplus 0 = 0$$

$$1 \oplus 1 = 0$$

$$1 \oplus 0 = 1$$

$$0 \oplus 1 = 1$$

$$1 \oplus 1 = 0$$

$$0 \oplus 0 = 0$$

$$25. \quad 00100101.$$



*Unit 5*

**INTRODUCTION  
TO PROGRAMMING**

## CONTENTS

Introduction .....	5-3
Unit Objectives .....	5-4
Unit Activity Guide .....	5-5
Branching .....	5-6
Conditional Branching .....	5-19
Algorithms .....	5-29
Additional Instructions .....	5-45
Experiments 7 and 8 .....	5-56
Unit Examination .....	5-57
Unit Examination Answers .....	5-60



## INTRODUCTION

In the final analysis, there are only two things you can do with a microprocessor: you can program it, and you can interface it with the outside world. Through programming, a microprocessor allows a robot to perform specific predetermined tasks. Also, because of its interfacing ability, the microprocessor allows a robot to establish communications and function with the outside world.

In this unit, you will learn to program the microprocessor. The written text, along with the associated experiments, will serve as an introduction to programming. In a later unit, you will learn how the robot, through the use of microprocessor control, is interfaced to the outside world.

The programs you encounter in this unit are simple enough that anyone can understand them, and yet, they illustrate many important concepts. By studying these programs, you will develop an understanding of how the microprocessor handles complex tasks. At the same time, you will gain practice using the instruction set.

## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Develop flow charts that illustrate step-by-step procedures for solving simple problems.
2. Explain the purpose of conditional and unconditional branching.
3. Using the block diagram of the hypothetical microprocessor, trace the data flow during the execution of a branch instruction.
4. Compute the proper relative address for branching forward or backward from one point to another in a program.
5. Explain the purpose of the carry, negative, zero, and overflow flags. Give an example of a situation that can cause each to be set and another example that will cause each to clear. List eight instructions that test one of these flags.
6. Write programs that can: multiply by repeated addition; divide by repeated subtraction; convert binary to BCD; convert BCD to binary; add multiple-precision numbers; subtract multiple-precision numbers; add BCD numbers.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the section on “Branching.”	_____
<input type="checkbox"/> Answer Programmed Review Questions 1 — 8.	_____
<input type="checkbox"/> Read the section on “Conditional Branching.”	_____
<input type="checkbox"/> Answer Programmed Review Questions 9 — 21.	_____
<input type="checkbox"/> Read the section on “Algorithms.”	_____
<input type="checkbox"/> Answer Programmed Review Questions 22 — 27.	_____
<input type="checkbox"/> Read the section on “Additional Instructions.”	_____
<input type="checkbox"/> Answer Programmed Review Questions 28 — 40.	_____
<input type="checkbox"/> Perform Experiments 7 and 8.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## BRANCHING

The programs discussed in the previous unit were all “straight line” programs; the instructions were executed one after another in the order in which they were written. Programs of this type are extremely limited because they use only a fraction of the microprocessor’s power.

The real power of the microprocessor comes from its ability to execute a section of a program over and over again. In an earlier program we saw that two numbers could be multiplied by repeated addition. As long as the numbers are very small and we know the value of the two numbers, we can write a “straight line” program to multiply the numbers. For example, 9 could be multiplied by 4 with the following program:

Address	Instruction/Data	Comments
00	LDA 05	Load direct
01	ADD 05	Add direct
02	ADD 05	Add direct
03	ADD 05	Add direct
04	HLT	
05	09	

This technique is very crude for a number of reasons. If the two numbers are large, such as 98 and 112, the number of ADD instructions becomes excessive. Moreover, the values of the two numbers to be multiplied are generally not known. Therefore, even if we were willing to write enough ADD instructions, we simply would not know how many to write. Obviously, some better technique must be available.

A technique that is used in virtually every program is called **looping**. This allows a section of the program to be run as often as needed. Every microprocessor has a group of instructions called JUMP or BRANCH instructions that allow it to execute these program loops. These allow the microprocessor to escape the normal instruction sequence.

The microprocessor discussed in this course has both jump and branch instructions. In this unit, we will confine our discussion to the branch instructions. In a later unit we will discuss the jump instructions.

Before discussing the types of branch instructions, we must first discuss a new addressing mode called relative addressing.

## Relative Addressing

Previously, we discussed immediate addressing and direct addressing. Recall that in the immediate addressing mode no address is specified. The data is assumed to be the byte following the opcode. In direct addressing, an address is given. The data is assumed to be at that address.

Branch instructions are somewhat different from the instructions discussed earlier. While the branch instruction has an address associated with it, the address does not indicate the location of data. Instead, the address indicates the location of the next instruction that is to be executed.

The format of the branch instruction is shown in Figure 5-1. All branch instructions are 2-byte instructions. The first byte is the 8-bit opcode. This code identifies the particular type of branch instruction. As you will see later, a microprocessor may have a dozen or more different branch instructions. Each has its own opcode that uniquely identifies it.

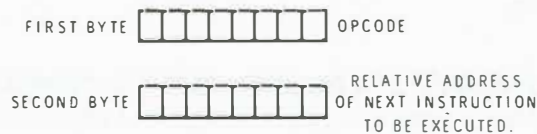


Figure 5-1  
Format of the branch instruction.

The second byte of the branch instruction indicates the point to which the program is to branch. That is, it specifies the address of the next instruction that is to be executed.

In some microprocessors, the address is absolute. That is, the address is the memory location that holds the next instruction. In this case, the instruction `BRANCH 3016` would mean that the instruction to be executed next is at address 30<sub>16</sub>. In other words, some microprocessors use direct addressing when branching.

Our hypothetical microprocessor uses a different technique called relative addressing. In this addressing mode, the byte following the opcode does not represent an absolute address. Instead, it is a number that must be added to the program counter to form the new address. Consider the instruction:

`BRANCH 3016`

Using relative addressing, this does not mean that the next instruction is to be taken from memory location  $30_{16}$ . Rather, it means that  $30_{16}$  must be added to the present contents of the program counter. Thus, if the program counter is at  $08_{16}$  when the BRANCH  $30_{16}$  instruction is executed, the next instruction will be fetched from location  $08_{16} + 30_{16} = 38_{16}$ .

By the same token, if the contents of the program counter is  $FA_{16}$  when a BRANCH  $03$  is encountered, the next instruction will be fetched from location  $FA_{16} + 03 = FD_{16}$ . Notice that this allows the MPU to jump over the instruction at addresses  $FB_{16}$  and  $FC_{16}$ .

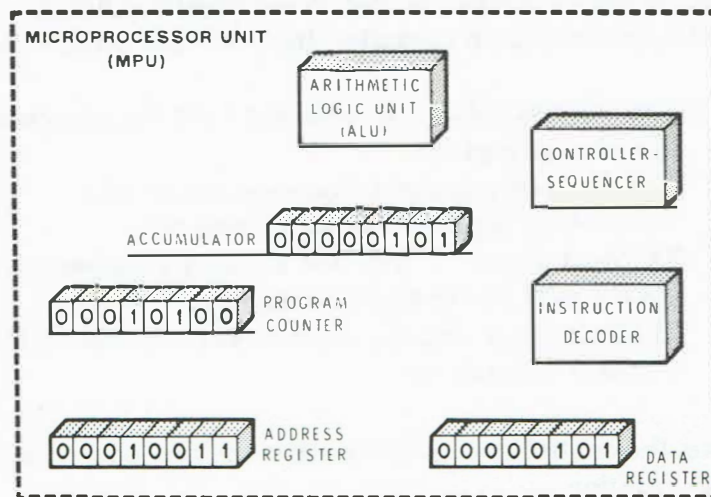
## Executing a Branch Instruction

Determining the relative address to use as the second byte of the branch instruction can be confusing unless you keep in mind the method by which the MPU executes a program. Therefore, let's go through the manipulations that take place within the MPU during the execution of the branch instruction.

Figure 5-2 shows sections of a program stored in memory. Let's assume that the MPU has been executing this program. Let's further assume that the MPU just completed the execution of the LDA  $05$  instruction at addresses  $12_{16}$  and  $13_{16}$ . The address register still holds the address of the last byte that was read from memory. The accumulator and data register hold the contents ( $05$ ) of the last location that was read out.

Notice that the program counter contains the address of the next instruction to be executed. This address points to the branch instruction in memory location  $14_{16}$ . Let's pick up the action at this point.





MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS
0001 0010	1000 0110	LDA
0001 0011	0000 0101	05 <sub>16</sub>
0001 0100	0010 0000	BRA
0001 0101	0000 0111	07 <sub>16</sub>
0001 0110	—	—
0001 1100	—	—
0001 1101	1000 1011	ADD
0001 1110	0000 0110	06 <sub>16</sub>
0001 1111	—	—

Figure 5-2  
Status of the MPU registers after executing  
the LDA 05 instruction.

Figure 5-3 shows how the first byte of the branch instruction is fetched. This is the standard fetch operation that was discussed earlier:

1. The address ( $14_{16}$ ) is transferred from the program counter to the address register.
2. The program counter is incremented to  $15_{16}$ .
3. The address is placed on the address bus.
4. The contents of the selected memory location are transferred via the data bus to the data register.
5. The instruction decoder examines this opcode and finds it to be a branch instruction.

Therefore, the controller-sequencer starts the procedure for executing a branch instruction.

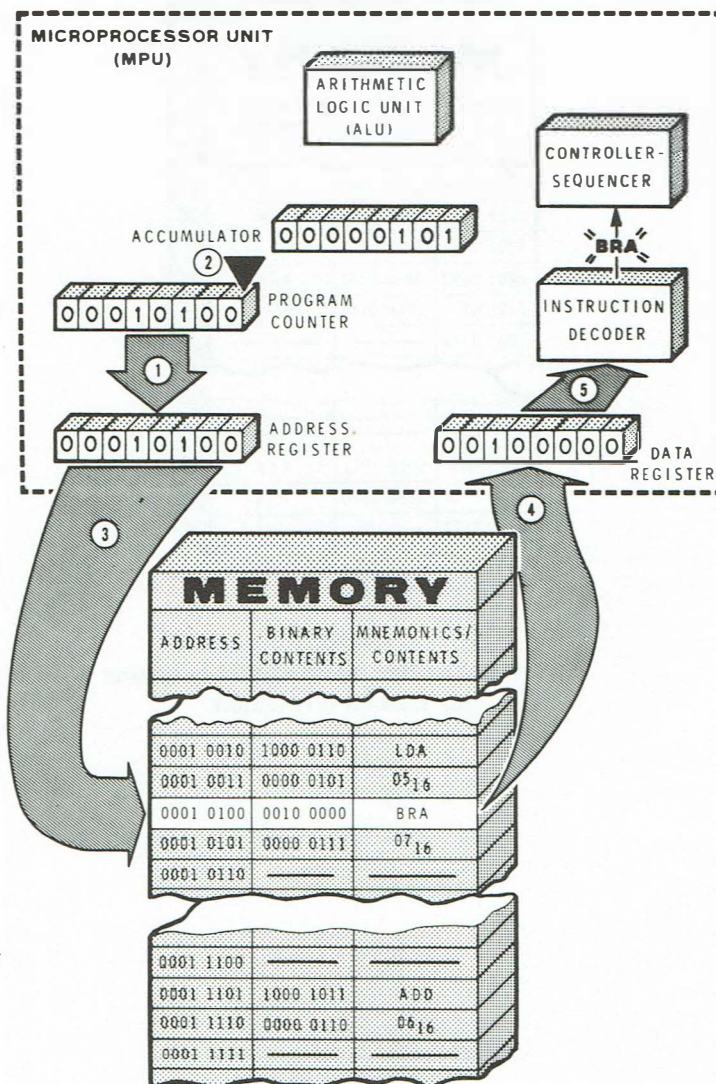


Figure 5-3  
Fetching the BRA instruction.

During the next machine cycle, the relative address is fetched. This procedure is shown in Figure 5-4. The major events are:

1. The address ( $15_{16}$ ) is transferred from the program counter to the address register.
2. The program counter is incremented to  $16_{16}$ .
3. The address ( $15_{16}$ ) is placed on the address bus.
4. The contents of location  $15_{16}$  are transferred to the data register via the data bus.

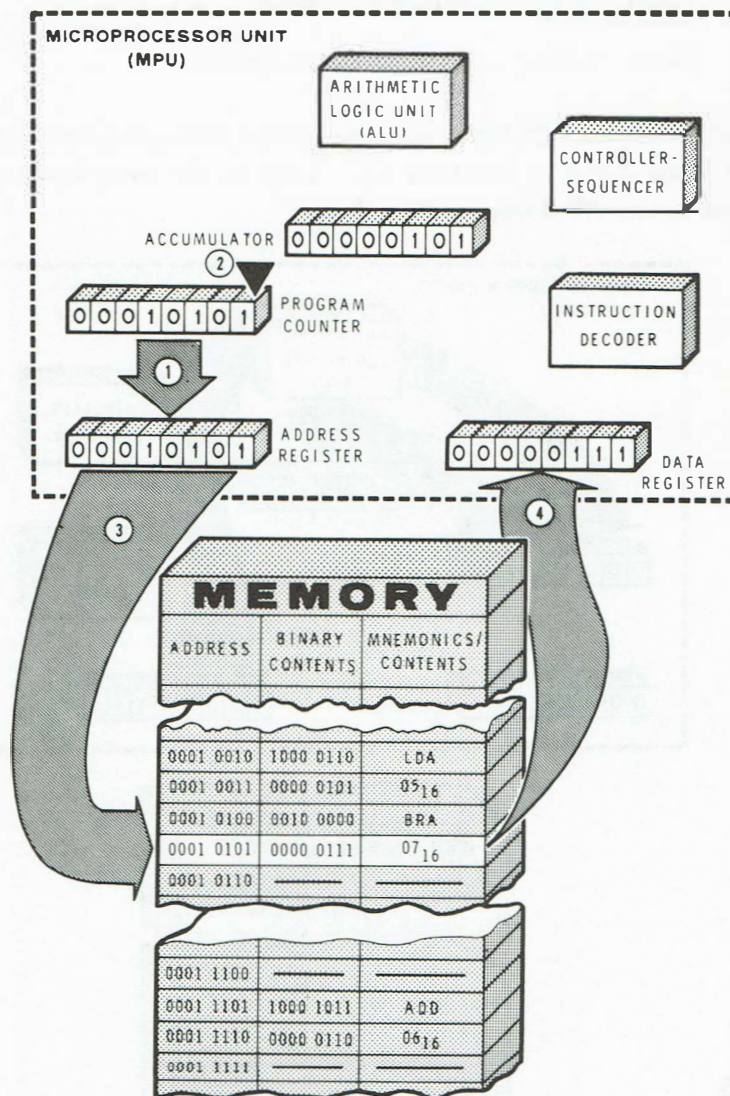


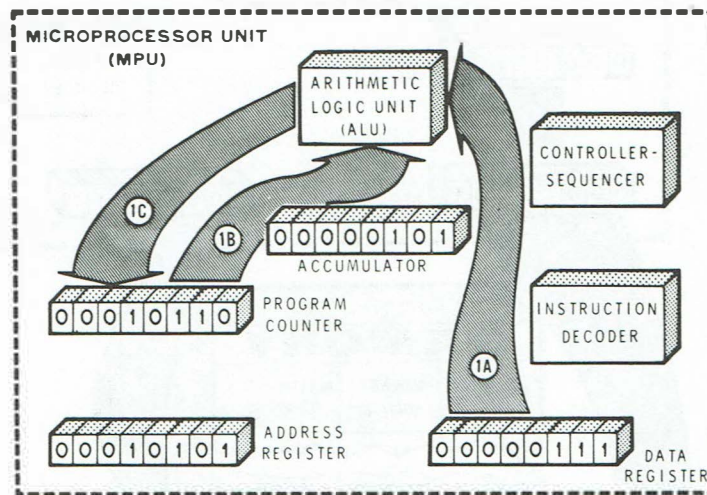
Figure 5-4  
Fetching the relative address.



Figure 5-5 shows the state of the various registers after the relative address is fetched. The relative address ( $07_{16}$ ) is in the data register. Now look at the program counter. Notice that it points to address  $16_{16}$ . However, the MPU has not yet finished executing the branch instruction. It must now compute the new address by adding the relative address to the program count. It uses the addition capabilities of the ALU to perform this function. That is, the program count and relative address are strobed into the ALU. The ALU adds the two together and produces a sum of

0001	0110	program count
0000	0111	relative address
<hr/>		
0001	1101	new program count

This sum is loaded into the program counter. Thus, the next instruction is fetched from memory location  $1D_{16}$ . That is, the next instruction to be executed is the  $\text{ADD } 06_{16}$  instruction.



MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS
0001 0010	1000 0110	LDA
0001 0011	0000 0101	$05_{16}$
0001 0100	0010 0000	BRA
0001 0101	0000 0111	$07_{16}$
0001 0110	-----	-----
0001 1100	-----	-----
0001 1101	1000 1011	ADD
0001 1110	0000 0110	$06_{16}$
0001 1111	-----	-----

Figure 5-5  
Computing the address of the next instruction.

## Branching Forward

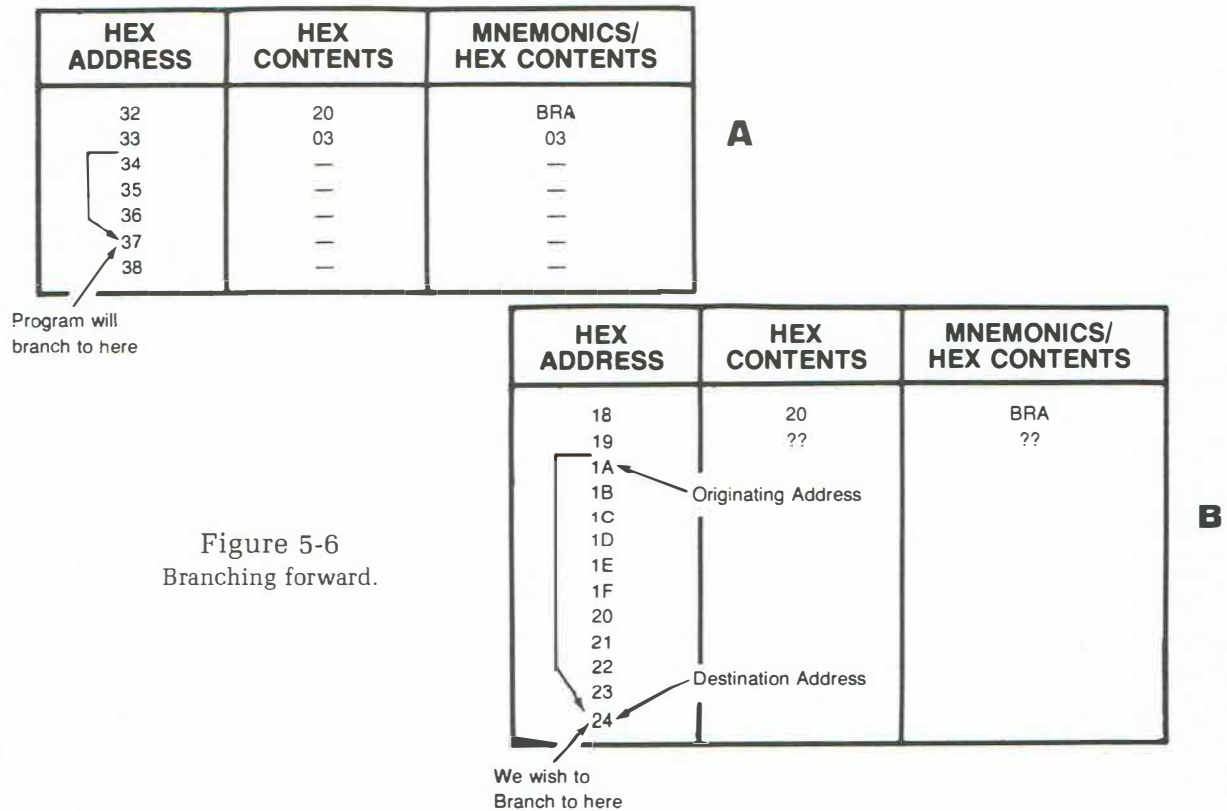
Branching in the forward direction is a simple task if you know the value of the program count when the relative address is added. A couple of examples will illustrate the procedure.

In Figure 5-6A, the BRANCH 03 instruction is placed in locations 32<sub>16</sub> and 33<sub>16</sub>. Assuming this instruction is executed, from which location will the next instruction be fetched? Remember that the program counter will always point to the next byte in sequence. Since the last byte fetched was the relative address from location 33<sub>16</sub>, the program counter must be at 34<sub>16</sub> when the relative address is added. Adding the relative address produces a new program count of

$$\begin{array}{r} 34_{16} \\ + 3_{16} \\ \hline 37_{16} \end{array}$$

Thus, the next instruction will be fetched from location 37<sub>16</sub>.

Figure 5-6B shows a slightly different situation. Here we wish to branch to the instruction at address 24<sub>16</sub>. The opcode for the branch instruction is at address 18<sub>16</sub>. What relative address is required at location 19<sub>16</sub> in order to implement this branch?



Keep in mind that the program count will automatically advance to  $1A_{16}$  after the relative address is fetched from address  $19_{16}$ . Also, remember that the relative address is added to the program count. Thus, a relative address of 00 would result in a "branch" to location  $1A_{16}$ . A relative address of 01 would result in a branch to location  $1B_{16}$ . Continuing this procedure until location  $24_{16}$  is reached, you find that a relative address of  $10_{10}$  is required. That is, the relative address must be  $0A_{16}$  or  $10_{10}$ .

There is a simple procedure for determining the relative address when branching forward. Subtract the originating address from the destination address. The difference is the relative address.

In our example, the originating address is  $1A_{16}$ . Remember this is the program count at the time the relative address is added. The destination address or the address to which you wish to branch is  $24_{16}$ . Subtracting the originating address from the destination address, you find that the required relative address is

$0010\ 0100_2$	$24_{16}$	Destination address
$-0001\ 1010_2$	$1A_{16}$	Originating address
$\hline 0000\ 1010_2$	$0A_{16}$	Relative address

As you can see, a relative address of  $0A_{16}$  is called for.



## Branching Backward

A backward branch is used when a part of the program is to be repeated. The technique used for branching backward is similar to that used in branching forward. The difference is that a negative number is used as the relative address. As you learned earlier, two's complement representation is used to signify negative and positive numbers. Therefore, the relative address portion of any branch instruction is interpreted as a two's complement number.

This means that bit 7 of the relative address byte is a sign bit. A 0 at bit 7 tells the MPU to branch forward; a 1 tells it to branch backward. Thus, the positive values for the relative address extend from  $0000\ 0000_2$  to  $0111\ 1111_2$ . This is  $00_{16}$  to  $7F_{16}$  and  $00_{10}$  to  $+127_{10}$ .

The negative values extend from  $1111\ 1111_2$  to  $1000\ 0000_2$ . This is  $FF_{16}$  to  $80_{16}$  and  $-1$  to  $-128_{10}$ . But remember, the relative address is with respect to the present program count. At the time the relative address is added, the program count points to the next byte after the relative address. Let's look at two examples of branching backward.

The first example is shown in Figure 5-7A. To what point does the MPU branch when the branch instruction at address  $5D_{16}$  is executed? Notice that the relative address is  $F9_{16}$ . In binary this is  $1111\ 1001_2$ . Recall that this is the two's complement representation of  $-7$ . Thus, the program count should jump backwards 7 bytes — but from what point? Recall that after the byte at address  $5E$  is fetched, the program count will automatically advance to  $5F_{16}$  or  $0101\ 1111_2$ . When the relative address ( $F9_{16}$  or  $1111\ 1001_2$ ) is added, the result is

0101 1111	← Old program count
+ 1111 1001	← Relative address
1 0101 1000	← New program count

↑

Carry is ignored

The carry bit is ignored, leaving a new program count of  $58_{16}$ . Thus, the next instruction will be fetched from address  $58_{16}$ .

Figure 5-7 shows a different problem. Here we want the branch instruction at addresses  $B0$  and  $B1$  to direct the MPU back to address  $A0$ . What relative address is required? A simple procedure is:

1. Subtract the destination address from the originating address.
2. Take the two's complement of the difference.

In our example, the program count will be advanced to  $B2_{16}$  after the relative address is fetched. This is our originating address. The point to which we wish to branch is  $A0_{16}$ . This is our destination address. Subtracting yields a difference of

1011 0010 <sub>2</sub>	$B2_{16}$	Originating address
– 1010 0000 <sub>2</sub>	$A0_{16}$	Destination address
0001 0010 <sub>2</sub>	$12_{16}$	Difference

Next you compute the relative address by taking the two's complement of the difference. The two's complement of  $0001\ 0010_2$  is  $1110\ 1110_2$ . In hexadecimal this is  $EE_{16}$ . Thus, the required relative address is  $EE_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56	—	—
57	—	—
58	—	—
59	—	—
5A	—	—
5B	—	—
5C	—	—
5D	20	BRA
5E	F9	F9
5F	—	—

Program branches to here

**A**

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
A0	—	—
A1	—	—
A2	—	—
A3	—	—
A4	—	—
A5	—	—
A6	—	—
A7	—	—
A8	—	—
A9	—	—
AA	—	—
AB	—	—
AC	—	—
AD	—	—
AE	—	—
AF	—	—
B0	20	BRA
B1	??	??
B2	—	—

We wish to branch to here

**B**

Figure 5-7  
Branching backwards.

## Programmed Review

1.	A technique called _____ allows a section of a program to be run as often as needed.
2.	(looping) All branch instructions are _____ byte instructions. (one/two)
3.	(two) When relative addressing is used by the branch instruction, the byte following the opcode _____ represent an absolute address. (does/does not)
4.	(does not) The _____ starts the procedure for executing a branch instruction.
5.	(controller-sequencer) A _____ two's complement number causes a branch forward. positive/negative
6.	(positive) A negative _____ complement number causes a branch backwards. (two's/ten's)
7.	(two's) A maximum of _____ memory locations (+127 <sub>10</sub> /+128 <sub>10</sub> ) can be branched over during a forward branch.
8.	(+127 <sub>10</sub> ) A maximum of _____ memory locations (-127 <sub>10</sub> /-128 <sub>10</sub> ) can be branched over during a backwards branch.
	(-128 <sub>10</sub> )

## CONDITIONAL BRANCHING

The branch instruction allows the MPU to jump forward over a block of data or over a portion of a program. It also allows the MPU to jump backwards so a group of instructions can be repeated.

Until now we have been discussing the **unconditional** branch instruction. This type of instruction always results in a program branch. For this reason, it is called the **BR**anch **A**lways instruction. Its mnemonic is BRA.

There are other types of branch instructions that greatly expand the versatility of the MPU. These are called **conditional** branch instructions. Unlike BRA, these instructions cause a branch only if some specified condition is met.

A good example of a conditional branch instruction is the Branch If Minus (BMI). This instruction may or may not initiate a branch operation, depending on the result of some previous arithmetic or logic operation. This instruction might be placed after a subtract instruction. If the result of the subtraction is a negative number, the branch would be implemented. Otherwise, the MPU would continue to fetch and execute instructions in numerical order. An example may help to illustrate this.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
95	96	LDA	Load accumulator direct
96	B0	B0	with contents of this address.
97	90	SUB	Subtract
98	B1	B1	the contents of this address.
99	2B	BMI	If result is minus
9A	03	03	branch this far.
9B	97	STA	If result is not minus, store
9C	B2	B2	at this address;
9D	3E	HLT	then halt.
9E	97	STA	If result is minus, store
9F	B3	B3	it at this address;
A0	3E	HLT	then halt.

Figure 5-8

This program uses the BMI instruction  
to make a simple decision.

Figure 5-8 shows part of a program that uses the branch if minus (BMI) instruction. Let's start with the instruction at address 95<sub>16</sub>. This instruction causes the contents of location B0<sub>16</sub> to be loaded into the accumulator. Next, the SUB instruction subtracts the contents of location B1<sub>16</sub> from the number in the accumulator. The next instruction (BMI) examines the result of the subtraction. If the result was a minus number, the program will branch over the next three bytes. That is, the next instruction to be executed is the STA instruction at address 9E<sub>16</sub>. Thus, the resulting number in the accumulator is stored in location B3<sub>16</sub> and the MPU halts.

If the result of the subtraction is not minus, the BMI instruction has no effect. That is, the BMI instruction is fetched and executed but no branch occurs because the specified condition is not met. In this case, the next instruction to be executed is the STA instruction at address  $9B_{16}$ . Thus, the result of the subtraction will be stored in location  $B2_{16}$ .

Notice that the program flow can take one of two paths, depending on the result of the subtraction. The BMI instruction gives the MPU this capability. The conditional branch instructions are sometimes called “decision making instructions.” The reason for this becomes obvious if you consider the implications of our sample program. Here the MPU decides if the number at address  $B1_{16}$  is larger than that at  $B0_{16}$ . The program path is determined by the outcome of this decision. If the number in  $B1_{16}$  is larger, the result of the subtraction is a negative number. In this case, the result is stored in location  $B3_{16}$ . Otherwise, the resulting difference is stored in location  $B2_{16}$ .

Virtually all programs must make some type of decision. Some frequently encountered decisions are:

“Which of two numbers is larger?”

“Does this byte represent a letter of the alphabet or a numeral?”

“Are these two numbers equal?”

“Is this an even number?”

“Has the program loop been repeated the proper number of times?”

Conditional branch instructions are used in making all of these decisions.



## Condition Codes

As the name implies, a conditional branch instruction causes a program branch only if some specified condition is met. Some commonly monitored conditions are:

1. Did a previous operation result in a negative number in the accumulator?
2. Did a previous operation result in zero in the accumulator?
3. Did a previous operation result in a carry from bit 7 of the accumulator?

To keep track of these conditions, most microprocessors have a group of single bit registers called condition code registers. Three of these registers are shown in Figure 5-9. They are the Negative (N) Register, the Zero (Z) Register, and the Carry (C) Register.

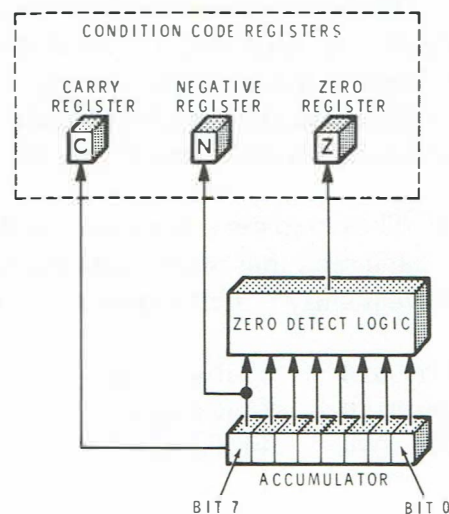


Figure 5-9  
Condition code registers monitor the operations in the accumulator.

**Negative (N) Register** Recall that negative numbers are expressed in two's complement form. Using this system, the most significant bit determines whether or not the number is negative. In an 8-bit byte, bit 7 is a 1 if the two's complement number is negative. Thus, the N register monitors bit 7 of the accumulator. Immediately after an operation that involves the accumulator, the N register looks at bit 7 to see if the number is negative. If so, the N register is set to 1. If the number in the accumulator is not negative, the N register is reset to 0.

Most operations that involve the accumulator affect the N register in this way — **but not all**. In a later unit we will point out how this register is affected by each instruction. In this unit, we will assume that the N register is affected as outlined above any time a number is added to, subtracted from, loaded into, or stored from the accumulator.

Another name for a condition code is a flag. Thus, the N register is sometimes called the N flag or the negative flag.

**Zero (Z) Register** This register monitors the accumulator looking for all zeros. Immediately after an operation that involves the accumulator, the zero-detect circuit looks at the resulting number. If all 8 bits are 0, the Z register is set to 1. Otherwise, the Z register is reset to 0. Most operations that involve the accumulator affect the Z register in this way.

**Carry (C) Register** The C register acts somewhat like an extension of the accumulator. You have seen that when two unsigned 8-bit numbers are added, the sum is frequently a 9-bit number. For example:

1001 0010	8-bit addend
+ 1100 0110	8-bit augend
<hr/>	
1 0101 1000	9-bit sum
↑	
carry	

Since the accumulator is an 8-bit register, the sum will not fit. The most significant bit (the carry) would be lost if you did not have another 1-bit register to hold it. This is the purpose of the C register. Any operation that causes a carry out of bit 7 will set the carry register to 1. Arithmetic operations that do not result in a carry will reset this register to 0.

The carry register is also used to keep track of “borrows” during subtract operations. If a subtraction requires a borrow for bit 7, the carry flag will also be set. For example, suppose you subtract an unsigned, binary number from a smaller unsigned binary number. The result will, of course, be a negative number. Moreover, bit 7 will have to “borrow” a bit to complete the subtraction. As a simple example, let’s subtract 2 from 1. The subtraction looks like this

Borrow →	1	
	0000 0001	Minuend
	– 0000 0010	Subtrahend
	<u>1111 1111</u>	<u>Difference</u>

The carry bit is set to 1 to indicate that a borrow operation occurred. Many subtraction operations do not require borrows. In these cases, the carry bit is reset to 0 to indicate that no borrow occurred.

Notice that the carry code can have different meanings, depending on the operation involved. That is, a 1 can mean either that a carry occurred or that a borrow occurred. The precise meaning of the 1 depends on whether the operation was an addition or a subtraction. We will discuss some additional aspects of the carry register in a later unit.

**Overflow (V) Register** The final condition code that is to be considered in this unit keeps track of two's complement overflow. Figure 5-10 shows how this register is connected in the MPU. A special circuit detects an overflow condition by monitoring bit 7 of the ALU's input and output lines. This circuit sets the V flag when an overflow occurs but clears it if no overflow occurs.

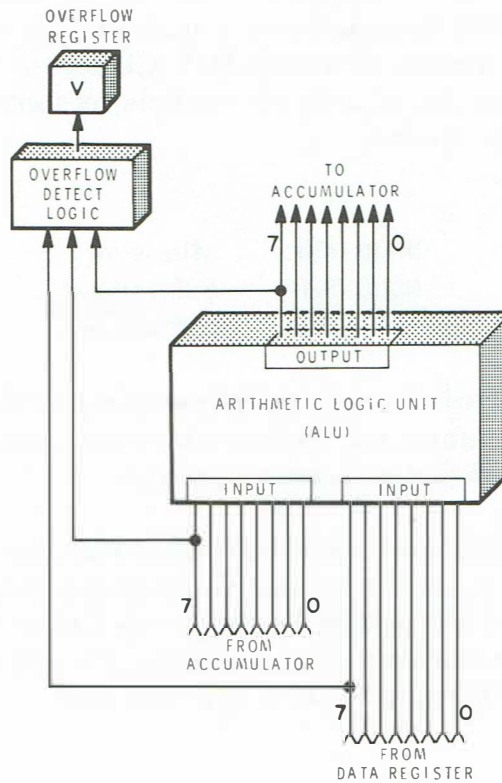


Figure 5-10  
The overflow register monitors bit 7  
of the ALU's input and output lines.

Let's see what is meant by two's complement overflow. Recall that the ALU adds numbers as if they were unsigned binary numbers. Even so, it can handle signed binary numbers if the proper bit patterns represent the negative numbers. This is the reason that the two's complement method of representing signed numbers has become so popular. A disadvantage of this system is that the magnitude of the number must be represented by 7 bits, since the eighth bit is used as the sign. Remember that a 1 in the MSB defines the number as negative.

Unfortunately, if two signed numbers are added and their sum exceeds 7-bits, the sign bit will be changed. For example, assume that a program adds  $+73_{10}$  and  $+96_{10}$ . The addition looks like this:

$$\begin{array}{r} \underline{0100\ 1001_2} \quad +73_{10} \\ \underline{0110\ 0000_2} \quad +96_{10} \\ 1010\ 1001_2 \quad 169_{10} \end{array}$$

The answer is correct if all the binary numbers represent unsigned quantities. However, using two's complement, the underlined bits represent sign bits. Therefore, the answer does **not** represent  $169_{10}$ . Instead, it represents  $-87_{10}$ . The reason for this error is that there was an overflow from bit 6 into the sign bit (bit 7). This is one of the situations that the V flag indicates.

When two's complement numbers having the same sign are added, the sum should have the same sign. That is, when two positive numbers are added, the sum should be positive. By the same token, when two negative numbers are added, the sum should be negative. However, an overflow will cause the sign to be reversed. The overflow logic detects this situation and sets the V flag whenever an overflow occurs.

The sign bit can also be upset during subtract operations. For example, when a negative number is subtracted from a positive number, the results should be positive. Remember that subtracting a negative number is the same as adding a positive number. However, in certain cases, an overflow can reverse the sign bit. This type of overflow occurs when the signs of the minuend and subtrahend are opposite and the difference has the sign of the subtrahend. This condition also sets the V flag.

## Conditional Branch Instructions

The conditional branch instructions available in our hypothetical microprocessor are shown in Figure 5-11. While these are largely self-explanatory, two points should be mentioned.

The first instruction, Branch If Carry Clear (BCC), monitors the C register. If the carry register is reset to 0, the branch is implemented. Notice that the words “clear” and “reset” are used interchangeably in this regard. They both mean the register contains a 0.

The branch instructions that monitor the Z register can also be confusing. The Branch If Equal Zero (BEQ) instruction implements a branch when the Z register is set to 1. Recall that the Z register is set to 1 when the number in the accumulator is zero. Thus, you must remember that a 0 in the Z register means that the number in the accumulator is **not** zero.

These conditional branch instructions can be used with other instructions to make a wide range of decisions. They greatly increase the power of the microprocessor. More than any other type of instruction, the conditional branches are responsible for the MPU’s “intelligence.” In the next section, you will see how these instructions are used.

INSTRUCTION	MNEMONIC	OPCODE	BRANCH IF
Branch If Carry Clear	BCC	24	C=0
Branch If Carry Set	BCS	25	C=1
Branch If Not Equal Zero	BNE	26	Z=0
Branch If Equal Zero	BEQ	27	Z=1
Branch If Plus	BPL	2A	N=0
Branch If Minus	BMI	2B	N=1
Branch If Overflow Clear	BVC	28	V=0
Branch If Overflow Set	BVS	29	V=1

Figure 5-11  
Conditional Branch Instructions.



## Programmed Review

9.	The _____ (conditional/unconditional)	branch instruction always results in a program branch.
10.	(unconditional)	The Branch If Minus (BMI) instruction is an example of a/an _____ (conditional/unconditional) branch instruction.
11.	(conditional)	The (BMI) instruction is used to test the _____ flag to see if it is set.
12.	(Negative (N))	In an 8-bit byte, the (N) register monitors bit _____ of the accumulator. (0/7)
13.	(7)	Generally speaking, the N flag is said to be set if the previous instruction left a _____ in the MSB of the accumulator. (0/1)
14.	(1)	The (Z) register monitors the accumulator immediately _____ an operation involving the accumulator. (before/after)
15.	(after)	If the previous instruction left the number $00010000_2$ in the accumulator, the (Z) flag would be set to _____. (0/1)
(cont'd.)		

16. (0) During the add operation, the \_\_\_\_\_ flag is set if there is a carry from bit 7 of the accumulator.

17. (carry (C)) During a subtract operation, the C flag is set if bit 7 had to \_\_\_\_\_ a bit to complete the subtraction.

18. (borrow) The C flag \_\_\_\_\_ represent both carry and borrow functions. (can/cannot)

19. (can) The \_\_\_\_\_ condition code register is used to keep track of two's complement overflow.

20. (Overflow (V)) The Branch If Equal Zero (BEQ) instruction causes a branch to occur only if the \_\_\_\_\_ register is set to 1.

21. (Z) If the N register is clear, the Branch If Plus (BPL) instruction \_\_\_\_\_ cause a branch to occur. (will/will not)

(will)

## ALGORITHMS

An algorithm is a step-by-step procedure for doing a particular job. It generally involves doing a complex task by stringing together a series of simple steps. To illustrate the use of an algorithm, consider the following very simple example.

### Multiplying by Repeated Addition

Most microprocessors do not have hardware multiply capabilities. That is, they do not have a multiplication circuit nor a multiply instruction. Nevertheless, the microprocessor can be made to multiply by use of an algorithm. One procedure for doing this was discussed earlier. It involved adding the multiplicand to itself the number of times indicated by the multiplier. In the previous example, this was done by using a separate ADD instruction for each addition. This procedure is unsatisfactory for two reasons. First, it results in excessively long programs. Second, you must know the value of the multiplier so that you know how many ADD instructions to include.

A better approach, although still far from ideal, is to use a program loop that will multiply two numbers by repeated addition. For the time being, assume that the two numbers are both positive and that the product does not exceed  $255_{10}$ . Let's further assume that we use only the instructions which have been discussed up to this point. In fact, we will restrict ourselves to the instructions shown in Figure 5-12.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch If Carry Set	BCS			25	
Branch if Equal Zero	BEQ			27	
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure 5-12  
Instructions to be used.

The algorithm for multiplying by repeated addition is quite simple. To multiply A times B, you merely add A to a specific location B times. For example, to multiply 5 times 3, you clear a location and then add 5. You continue the addition until 5 has been added 3 times. The number in the affected location will then be  $15_{10}$  which is the product of 5 times 3.

The success of this operation depends on the microprocessor knowing when to stop. It must add 5 three times, but only three times. One way to keep track of the number of additions is to decrement the multiplier (3) each time an addition is made. When the multiplier reaches 0, the proper number of multiplications has been carried out.

Figure 5-13 is a flow chart that illustrates the algorithm. In the first two steps, the MPU clears the accumulator and stores the resulting number (0) in the product. This ensures that the product is zero before the first number is added. Next, it loads the multiplier and checks to see if the multiplier is 0. If so, the process is stopped since a multiplier of 0 dictates a product of 0.

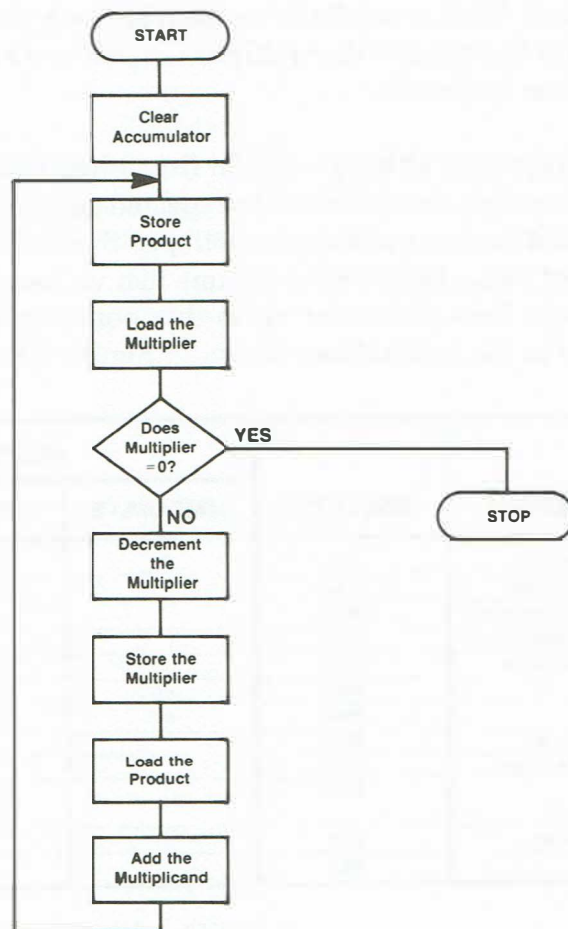


Figure 5-13  
Flow chart for multiplying by repeated addition.

In our example, the multiplier is 3; therefore, we exit the decision block via the “no” line. The next step tells us to decrement the multiplier. The new value of the multiplier (2) is stored for future use. Next, the product whose present value is 0 is loaded. Then, the multiplicand (5) is added so that the new value of the product becomes 5. This completes our first pass through the program. Remember that the multiplicand has been added once and that the multiplier has been reduced by one.

Notice that the program loops back to the input of the second block. The product which now has a value of 5 is stored back in memory. The multiplier (which is now 2) is loaded and tested. Because its value is not yet 0, the multiplier is decremented to 1 and stored again. The product (whose value is now 5) is then loaded and the multiplicand is added so that a new value of  $10_{10}$  is obtained.

The program loops again and the new product ( $10_{10}$ ) is stored. The multiplier (whose value is now 1) is loaded and tested. Because its value is still not 0, it is decremented again. Notice that the value of the multiplier is now 0. This value is stored away, the product ( $10_{10}$ ) is fetched, and the multiplicand is added once more. The new value of the product becomes  $15_{10}$ .

The program loops again and the product is stored. The multiplier is loaded and tested. Recall that the value of the multiplier is now 0. Consequently, we exit the decision block via the “yes” line. The program has accomplished its task and it now stops. Notice that the value of the product is  $15_{10}$  which is the proper answer for  $5 \times 3$ .

The next task is to convert the flow chart to a program that the computer can execute. Figure 5-14 shows such a program. Carefully compare this program to the flow chart paying particular attention to the comments column. Work through the program on paper and verify that it will multiply the numbers at addresses  $11_{16}$  and  $12_{16}$ . Although 3 and 5 are used in this example, the program will work for any values of multiplier and multiplicand as long as the product does not exceed  $255_{10}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA 13	Store the product in location $13_{16}$ .
02	13	LDA 12	Load the accumulator with the multiplier from location $12_{16}$ .
03	96	BEQ 09	If the multiplier is equal to zero, branch down to the Halt instruction.
04	12	DECA	Otherwise, decrement the multiplier.
05	27	STA 12	Store the new value of the multiplier back in location $12_{16}$ .
06	09	LDA 13	Load the accumulator with the product from location $13_{16}$ .
07	4A	ADD 11	Add the multiplicand to the product.
08	97	BRA F1	Branch back to instruction in location 01.
09	12	HLT	Halt.
0A	96		Multiplicand.
0B	13		Multiplier.
0C	9B		Product.
0D	11		
0E	20		
0F	F1		
10	3E		
11	05		
12	03		
13	—		

Figure 5-14

This program multiplies the numbers at addresses  $11_{16}$  and  $12_{16}$ , and places their product at address  $13_{16}$ .



## Dividing by Repeated Subtraction

Another interesting algorithm is one that allows the microprocessor to divide by repeated subtraction. The technique is to keep track of the number of times that the divisor can be subtracted from the dividend. For example, suppose you wish to divide  $47_{10}$  by  $15_{10}$ . The divisor can be subtracted 3 times:

First Subtraction      Second Subtraction      Third Subtraction

$$\begin{array}{r}
 47_{10} \\
 -15_{10} \\
 \hline
 32_{10}
 \end{array}
 \quad \xrightarrow{\quad} \quad
 \begin{array}{r}
 32_{10} \\
 -15_{10} \\
 \hline
 17_{10}
 \end{array}
 \quad \xrightarrow{\quad} \quad
 \begin{array}{r}
 17_{10} \\
 -15_{10} \\
 \hline
 2_{10}
 \end{array}$$

Because three subtractions occurred, the quotient is 3. Also, because 2 was left after the last subtraction, the remainder is 2. We can verify this by long division:

$$\begin{array}{r}
 \text{divisor} \quad \rightarrow \quad 15_{10} \overline{) 47_{10}} \\
 \underline{45_{10}} \\
 2_{10}
 \end{array}
 \begin{array}{l}
 3_{10} \leftarrow \text{Quotient} \\
 47_{10} \leftarrow \text{Dividend} \\
 2_{10} \leftarrow \text{Remainder}
 \end{array}$$

The microprocessor keeps track of the number of subtractions by incrementing the quotient by one each time a subtraction occurs. Of course, the quotient must be initially set to zero.

The divisor is subtracted from the dividend until any further subtraction would result in a negative number. The MPU can use the BMI instruction to check for a negative result on each loop. The negative result is the indication that the process is finished.

A flow chart for this algorithm is shown in Figure 5-15. The actual program is shown in Figure 5-16. The program is arbitrarily placed in locations 00 through 10<sub>16</sub>. The dividend (47<sub>10</sub>) is at address 11<sub>16</sub> while the divisor (15<sub>10</sub>) is at address 12<sub>16</sub>. When executed, the program will produce the quotient at location 13<sub>16</sub> and the remainder at location 11<sub>16</sub>.

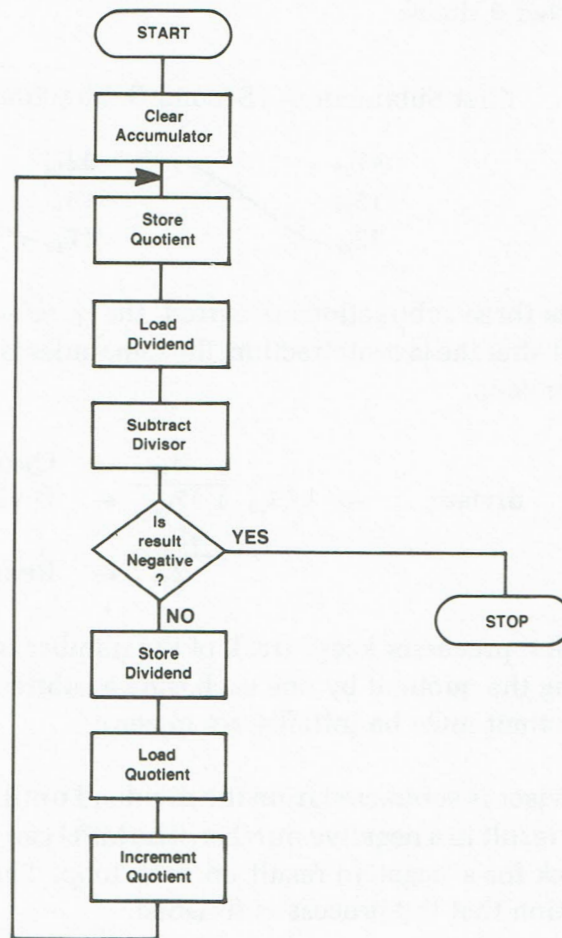


Figure 5-15

Flow chart for dividing by repeated subtraction.



HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store in the quotient which is at address location 13 <sub>16</sub> .
02	13	LDA	Load the accumulator with the dividend from location 11 <sub>16</sub> .
03	96	SUB	Subtract the divisor from the dividend.
04	11	BMI	If the difference is negative, branch down to the Halt instruction.
05	90	STA	Otherwise, store the difference back in location 11 <sub>16</sub> .
06	12	LDA	Load the accumulator with the quotient.
07	2B	INCA	Increment the quotient by one.
08	07	BRA	Branch back to instruction in location 01.
09	97	F1	Halt.
0A	11	HLT	
0B	96	2F	Dividend (47 <sub>10</sub> ).
0C	13	0F	Divisor (15 <sub>10</sub> ).
0D	4C	—	Quotient.
0E	20		
0F	F1		
10	3E		
11	2F		
12	0F		
13	—		

Figure 5-16

This program divides by repeatedly subtracting the divisor from the dividend.

Refer to the flow chart and the comments column of the program. Before reading further, try running through the program on paper. This will give you a feel for how the computer solves the problem.

Now let's go through the program to see what it does. The first two instructions clear the quotient. Next, the dividend (47<sub>10</sub>) is loaded into the accumulator and the divisor (15<sub>10</sub>) is subtracted. The BMI instruction is used to examine the difference (32<sub>10</sub>). Since the difference is not minus, the branch does not occur. Consequently, the next instruction stores the difference (32<sub>10</sub>) back in the location from which the dividend came. In effect, the difference becomes the new dividend. Next the quotient (0) is loaded and is incremented to 1. The program then branches back to the instruction in location 01. This instruction stores the quotient (1) back in location 13<sub>16</sub>.

On the next pass through the program, the new dividend (32<sub>10</sub>) is loaded and the divisor (15<sub>10</sub>) is subtracted again. This produces a difference of (17<sub>10</sub>). Since the difference is not negative, the BMI instruction does not cause a branch. Thus, the difference is stored back in location 11<sub>16</sub>. The quotient is loaded into the accumulator and is incremented to 2. The BRA instruction causes the program to loop once again. The STA instruction in location 01 stores the quotient (2) back in location 13<sub>16</sub>.

On the third pass the dividend ( $17_{10}$ ) is loaded and the divisor ( $15_{10}$ ) is subtracted a third time. The difference (02) is still not negative so no branch occurs. The difference is stored away; the quotient is loaded and is incremented to 03. Notice that this is the proper final value for the quotient. Therefore, on the next pass, the MPU should be able to break out of the loop.

The quotient is stored back in location  $13_{16}$ . The dividend, which now has a value of 2, is loaded. The divisor ( $15_{10}$ ) is subtracted, leaving a negative number ( $-13_{10}$ ) in the accumulator. The BMI instruction recognizes that this is a negative number and implements a branch operation. Notice that the MPU branches forward to the HLT instruction. Thus, the program ends with the quotient set to 3. The remainder is at address  $11_{16}$ . That is, the remainder is what remains of the dividend after the third subtraction.

It may bother you that there were four subtractions and that a negative difference resulted from the last subtraction. However, you will recall that the quotient was incremented only on the first three of these subtractions. Thus, the final quotient is 3. Moreover, the negative difference that resulted during the last subtraction was never stored. Consequently, the remainder was 2 when the program ended.

This program does have some drawbacks. For one thing, neither the dividend nor the divisor can exceed  $127_{10}$ . Also, only positive numbers can be used. Finally, the program gets hung up in an endless loop if the initial value of the divisor is zero. While division by zero is not allowed in mathematics, some provision would be made in a practical program to recognize this eventually. Since this program is only for demonstration purposes, we will live with these shortcomings for the time being.



## Converting BCD to Binary

When a microprocessor is used with a terminal such as a teletypewriter, numerals are entered as ASCII characters. For example, the number  $237_{10}$  is entered into memory as three ASCII characters:

Numeral	ASCII Character
2	0011 0010
3	0011 0011
7	0011 0111

Notice that the four least significant bits of the ASCII character represent the BCD value of the corresponding numeral. Thus, we can convert these ASCII characters to BCD numbers simply by eliminating the four most significant bits.

While the microprocessor does have some BCD capability, it is often desirable to convert BCD numbers to binary. The technique for doing this illustrates another useful algorithm.

The BCD representation for  $237_{10}$  is:

0010	←	Hundreds BCD digit
0011	←	Tens BCD digit
0111	←	Units BCD digit

Notice that in this example 0010 represents two hundred, 0011 represents thirty, and 0111 represents seven. Because of this, there is a simple procedure for converting BCD to binary. Starting with an initial value of zero, the MPU adds  $100_{10}$  as many times as indicated by the hundreds digit. It then adds  $10_{10}$  as indicated by the tens digit. Finally, the value of the units digit is added on to the result. The steps involved look like this:

1100100 <sub>2</sub>	100 <sub>10</sub>	One hundred added
1100100 <sub>2</sub>	100 <sub>10</sub>	2 times
1010 <sub>2</sub>	10 <sub>10</sub>	
1010 <sub>2</sub>	10 <sub>10</sub>	Ten added three times
1010 <sub>2</sub>	10 <sub>10</sub>	
0111 <sub>2</sub>	7 <sub>10</sub>	7 units added
<u>11101101<sub>2</sub></u>	<u>= 237<sub>10</sub></u>	

As you can see, this procedure ends with the binary result of 1110 1101, which is the binary representation for  $237_{10}$ .

A flow chart for this procedure is shown in Figure 5-17. Here, the first step is to clear the binary result. We will be adding to this result, so it must start out at zero.

Next the program enters a loop in which it adds  $100_{10}$  to the binary result the number of times indicated by the hundreds digit of the BCD number. The hundreds digit is loaded and tested for zero. If it is not zero, the hundreds digit is decremented and stored back in memory. Then the binary result is loaded and  $100_{10}$  is added. The result is stored away and the loop is repeated. In our example, the hundreds digit was initially 2. Thus, this loop is repeated twice. The binary result will have the value  $1100\ 1000_2$  ( $200_{10}$ ) when the hundreds digit is reduced to zero. At that time, the program exits the decision block via the "yes" line and immediately encounters a second loop.

The second loop is exactly like the first except that  $10_{10}$  is added to the binary result each time the tens digit of the BCD number is decremented. Because the tens digit was initially 3, this loop is repeated three times. Ten is added to the binary result three times, bringing the result to  $1110\ 0110_2$  ( $230_{10}$ ). The program exits this loop via the "yes" line on the pass after the tens digit is reduced to zero.

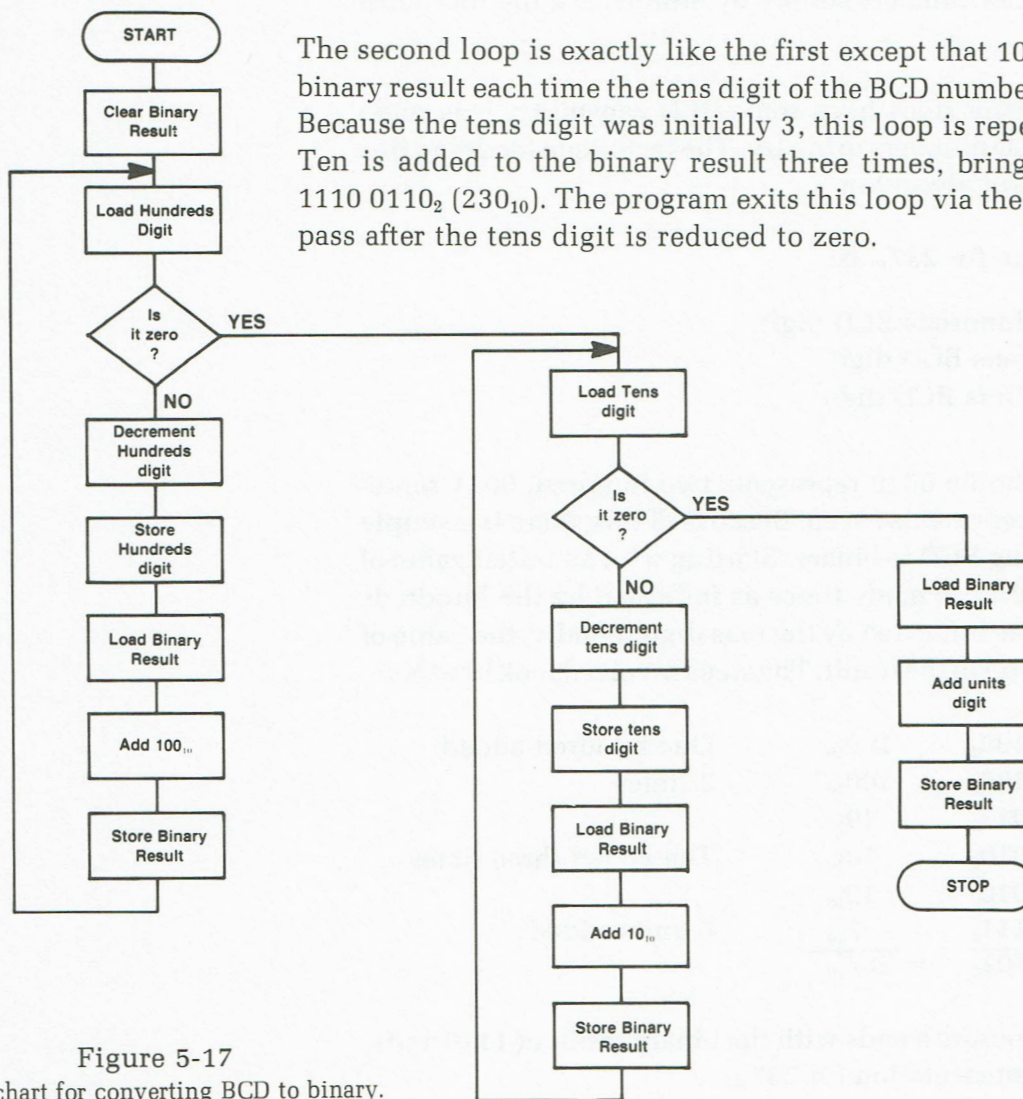


Figure 5-17

Flow chart for converting BCD to binary.



HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location 2B. This clears the binary result.
03	96	LDA	Load direct
04	28	28	the hundreds BCD digit.
05	27	BEQ	If the hundreds digit is zero, branch
06	0B	0B	forward to the instruction in location 12 <sub>16</sub> .
07	4A	DECA	Otherwise, decrement the accumulator.
08	97	STA	Store the result as the new
09	28	28	hundreds BCD digit.
0A	96	LDA	Load direct
0B	2B	2B	the binary result.
0C	8B	ADD	Add immediate
0D	64	64	100 <sub>10</sub> to the binary result.
0E	97	STA	Store away the new
0F	2B	2B	binary result.
10	20	BRA	Branch
11	F1	F1	back to the instruction in location 03 <sub>16</sub> .
12	96	LDA	Load direct
13	29	29	the tens BCD digit.
14	27	BEQ	If the tens BCD digit is zero, branch
15	0B	0B	forward to the instruction in location 21 <sub>16</sub> .
16	4A	DECA	Otherwise, decrement the accumulator.
17	97	STA	Store the result as the new
18	29	29	tens BCD digit.
19	96	LDA	Load direct
1A	2B	2B	the binary result.
1B	8B	ADD	Add immediate
1C	0A	0A	10 <sub>10</sub> to the binary result.
1D	97	STA	Store away the new
1E	2B	2B	binary result.
1F	20	BRA	Branch
20	F1	F1	back to the instruction in location 12 <sub>16</sub> .
21	96	LDA	Load direct
22	2B	2B	the binary result.
23	9B	ADD	Add direct
24	2A	2A	the units BCD digit.
25	97	STA	Store away the new
26	2B	2B	binary result.
27	3E	HLT	Halt.
28	02	02	Hundreds BCD digit.
29	03	03	Tens BCD digit.
2A	07	07	Unit BCD digit.
2B	—	—	Reserved for the binary result.

Figure 5-18

Program for converting BCD to binary.

The final three blocks add the units digit to the binary result. In our example, the units digit was 7<sub>10</sub>. This brings the final binary result to 1110 1101<sub>2</sub>. Notice that this is the proper binary representation for the unsigned number 237<sub>10</sub>.

A program that carries out this operation is shown in Figure 5-18. The three digit BCD number is stored in locations 28<sub>16</sub>, 29<sub>16</sub>, and 2A<sub>16</sub>. The binary equivalent will be computed and placed in location 2B<sub>16</sub>. Before reading further, try to work through the program. Refer to the flow chart and the comments column as you trace out the sequence that the MPU will follow.

Now let's briefly go through the program. The first two instructions clear the location at which the binary number will be formed.

Next, the program enters the first loop, which is shown at the first shaded area. In this loop, the hundreds digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary result is loaded and  $100_{10}$  is added. The result is stored away and the loop is repeated. Because the hundreds digit was 02 initially,  $100_{10}$  will be added to the binary result twice. Thus, upon leaving this loop, the binary result will have the value  $200_{10}$ . The MPU escapes this loop when the BEQ instruction at address 05 detects that the hundreds digit has been reduced to zero. The branch is to the second loop which is shown as an unshaded area.

In the second loop, the tens digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary number is loaded,  $10_{10}$  is added, and the result is stored away. This loop is repeated until the tens digit is reduced to zero. Because the tens digit was initially three, the loop is repeated three times so that thirty is added to the binary number. The BEQ instruction at address  $14_{16}$  allows the MPU to escape the loop and branch to the final program segment.

This final segment is the last shaded area. Here, the binary result is loaded and the units digit is added. This brings the binary result to  $237_{10}$ . Then the result is stored and the program halts. While the number  $237_{10}$  was used in this example, the program will convert any BCD number between 000 and  $255_{10}$  to its binary equivalent.



## Converting Binary to BCD

A microprocessor generally manipulates data in the form of straight binary numbers. However, before the results can be transmitted to the outside world, the data is often converted back to BCD. Frequently, this is an intermediate step in converting back to ASCII.

The binary-to-BCD conversion is the reverse of the process that occurred in the previous program. The MPU must determine how many times  $100_{10}$  can be subtracted from the binary number. The answer becomes the hundreds BCD digit. After the  $100_{10}$  has been subtracted as many times as possible,  $10_{10}$  is subtracted repeatedly from the remaining number. The number of subtractions becomes the tens BCD digit. Finally, after  $10_{10}$  has been subtracted as many times as possible, the remaining number becomes the units BCD digit.

For the number  $1110\ 1101_2$  ( $237_{10}$ ), the process looks like this:

1110 1101	237
<u>-0110 0100</u>	<u>-100</u>
1000 1001	137
<u>-0110 0100</u>	<u>-100</u>
0010 0101	37
<u>-0000 1010</u>	<u>-10</u>
0001 1011	27
<u>-0000 1010</u>	<u>-10</u>
0001 0001	17
<u>-0000 1010</u>	<u>-10</u>
0000 0111	7

hundreds digit = 2

tens digit = 3

units digit = 7

One hundred can be subtracted twice, resulting in the hundreds digit of  $2_{10}$  or  $0010_2$ . From the remainder, ten can be subtracted three times. Thus, the tens digit is  $3_{10}$  or  $0011_2$ . Finally, the remainder of  $7_{10}$  or  $0111_2$  becomes the units digit. The BCD representation is  $0010\ 0011\ 0111$ .

Figure 5-19 shows the flow chart for this procedure. The first three blocks clear the hundreds, tens, and units digits of the BCD result. Then the binary number that is to be converted to BCD is loaded and  $100_{10}$  is subtracted. The outcome is tested to see if a negative number resulted. If not, the result is stored away. The hundreds digit is loaded, incremented, and stored away. The loop is repeated until  $100_{10}$  can no longer be subtracted. In our example,  $100_{10}$  can be subtracted twice. Therefore, the hundreds digit is incremented to 2. The third subtraction of  $100_{10}$  gives a negative result. This allows the MPU to escape the first loop.

The second loop increments the tens digit to the proper value by subtracting  $10_{10}$  repeatedly while keeping track of the number of subtractions. In our example, this loop is repeated three times. Consequently, the tens digit is incremented to 3. The binary number that is left over after  $10_{10}$  is subtracted the proper number of times becomes the units digit. That is, upon escaping the second loop, the remaining binary number is stored in the units digit. In our example, the remaining number, and therefore, the units digit, is 7.

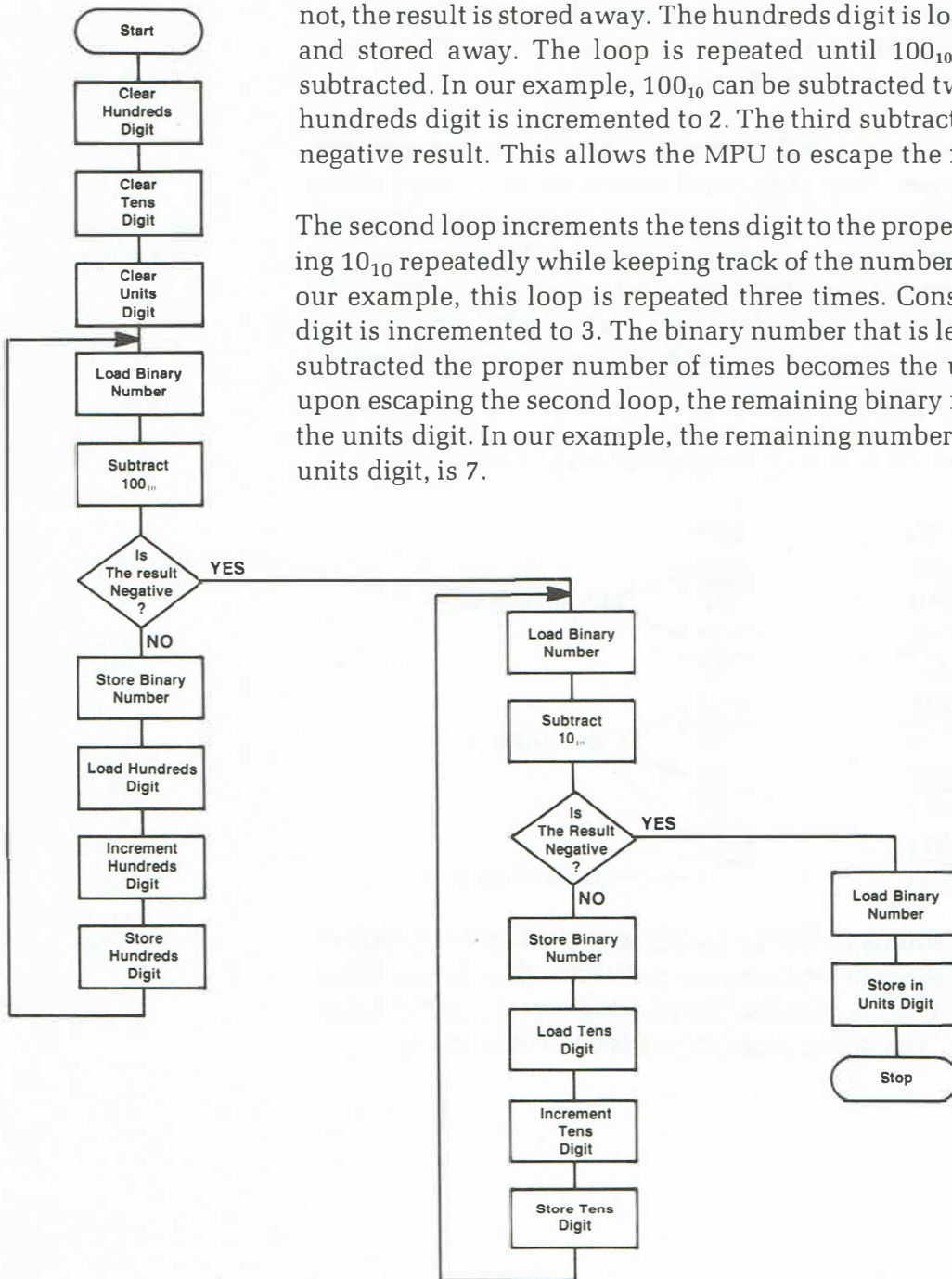


Figure 5-19  
Flow chart for converting a binary number  
to a BCD number.

The program that carries out this procedure is shown in Figure 5-20. At this point, you should be able to interpret the program from the comments given. However, a couple of points should be explained briefly. Any unsigned binary number from 0000 0000 to 1111 1111 can be placed at address  $2A_{16}$ . The computer will convert this number into its BCD equivalent. The hundreds digit will appear at address  $2B_{16}$ , the tens digit at  $2C_{16}$ , and the units digit at  $2D_{16}$ . The decision making instructions at addresses  $0B_{16}$  and  $1A_{16}$  are Branch if Carry Set (BCS) instructions. Because these instructions follow immediately after SUB instructions, the carry flag will indicate whether or not a borrow occurred. In effect, the BCS instructions decide: "Was the result of the subtraction a negative number?"

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location $2B_{16}$ . This clears the hundreds digit.
03	97	STA	Store 00
04	2C	2C	in location $2C_{16}$ . This clears the tens digit.
05	97	STA	Store 00
06	2D	2D	in location $2D_{16}$ . This clears the units digit.
07	96	LDA	Load direct
08	2A	2A	the binary number to be converted.
09	80	SUB	Subtract immediate
0A	64	64	$100_{10}$ .
0B	25	BCS	If a borrow occurred, branch
0C	09	09	forward to the instruction in location $16_{16}$ .
0D	97	STA	Otherwise, store the result of the subtraction
0E	2A	2A	as the new binary number.
0F	96	LDA	Load direct
10	2B	2B	the hundreds digit of the BCD result.
11	4C	INCA	Increment the hundreds digit.
12	97	STA	Store the hundreds digit
13	2B	2B	back where it came from.
14	20	BRA	Branch
15	F1	F1	back to the instruction at address $07_{16}$ .
16	96	LDA	Load direct
17	2A	2A	the binary number.
18	80	SUB	Subtract immediate
19	0A	0A	$10_{10}$ .
1A	25	BCS	If a borrow occurred, branch
1B	09	09	forward to the instruction in location $25_{16}$ .
1C	97	STA	Otherwise, store the result of the subtraction
1D	2A	2A	as the new binary number.
1E	96	LDA	Load direct
1F	2C	2C	the tens digit.
20	4C	INCA	Increment the tens digit.
21	97	STA	Store the tens digit
22	2C	2C	back where it came from.
23	20	BRA	Branch
24	F1	F1	back to the instruction at address $16_{16}$ .
25	96	LDA	Load direct
26	2A	2A	the binary number.
27	97	STA	Store it in
28	2D	2D	the units digit.
29	3E	HLT	Halt.
2A	—	—	Place binary number to be converted at this address.
2B	—	—	Hundreds digit
2C	—	—	Tens digit
2D	—	—	Units digit
			} Reserved for BCD result.

Figure 5-20  
Program for converting a binary number  
to a BCD number.



## Programmed Review

22. A step-by-step procedure for doing a particular job is called an \_\_\_\_\_.

23. (algorithm) Refer to the program in Figure 5-14. If the multiplier is  $8_{16}$  and the multiplicand is  $15_{16}$ , the BEQ instruction will be executed \_\_\_\_\_ times.

24. (nine) Using an algorithm that allows the microprocessor to divide by repeated subtraction, the microprocessor keeps track of the number of subtractions by \_\_\_\_\_ the quotient  
(incrementing/decrementing)  
by one each time a subtraction occurs.

25. (incrementing) When dividing by repeated subtraction, a \_\_\_\_\_ result indicates that the process is finished.  
(positive/negative)

26. (negative) When a microprocessor is used with a terminal such as a teletypewriter, numerals are usually entered as \_\_\_\_\_ characters.

27. (ASCII) You can convert ASCII characters to BCD numbers by eliminating the four \_\_\_\_\_ significant bits.  
(most/least)

(most)

## ADDITIONAL INSTRUCTIONS

Before leaving this unit, you should also look at four additional instructions. The names, opcodes and mnemonics of these instructions are shown in Figure 5-21.

NAME	MNEMONIC	HEX OPCODE		
		IMMEDIATE	DIRECT	INHERENT
ADD WITH CARRY	ADC	89	99	
SUBTRACT WITH CARRY	SBC	82	92	
ARITHMETIC SHIFT ACCUMULATOR LEFT	ASLA			48
DECIMAL ADJUST ACCUMULATOR	DAA			19

Figure 5-21  
Four new instructions.

Recall that the ALU always adds numbers as if they were unsigned binary numbers. When it adds 8-bit numbers, a carry often occurs from the MSB, setting the C flag. Thus, you can think of the carry flag as an extension of the accumulator. Let's look at some instructions that use the carry flag.

### Add With Carry (ADC) Instruction

This instruction is similar to the ADD instruction discussed earlier with one important difference. If the carry bit is set to 1 before this instruction is executed, 1 is added to the LSB of the sum. However, if the carry bit is 0 prior to execution, then no carry is added. The effect is the same as having the carry bit from the previous operation added to the result of the present operation.

Like the ADD instruction, the ADC instruction has two addressing modes: immediate and direct. As shown in Figure 5-21, the opcode for "ADD With Carry Immediate" is  $89_{16}$ , while the opcode for "ADD With Carry Direct" is  $99_{16}$ .



A primary use of the ADC instruction is to simplify **multiple-precision arithmetic**. Multiple-precision means that two or more bytes are used to represent a number. Recall that a single byte can represent unsigned binary numbers with values up to  $255_{10}$ . However, much larger numbers can be represented by using two or more bytes. Two bytes (16 bits) can represent unsigned binary values up to  $2^{16}-1$  or  $65,535_{10}$ . Three bytes can represent values to  $16,777,215_{10}$ ; etc. Thus, the MPU can handle numbers of virtually any size simply by stringing the proper number of bytes together.

Suppose, for example, that two very large numbers are to be added. Figure 5-22 shows how the addition might look on paper. Notice that two 24-bit numbers are being added to form a 24-bit sum. The MPU is restricted to operating on data in 8-bit bytes. Thus, each quantity involved must be represented by three bytes.

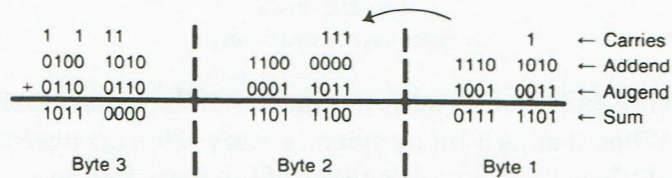


Figure 5-22  
Multiple-precision addition.

The MPU must be instructed to add the first byte of the addend to the first byte of the augend. This forms the first byte of the sum. Next the MPU must add the second bytes of the addend and augend. However, you will notice that there was a carry from the first byte to the second byte. If this carry is not added with the second bytes, the sum will be in error. The ADC instruction performs this operation automatically.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with
01	13	13	least significant byte of addend.
02	9B	ADD	Add direct
03	16	16	least significant byte of augend.
04	97	STA	Store result in
05	19	19	least significant byte of sum.
06	96	LDA	Load accumulator direct with
07	14	14	next byte of addend.
08	99	ADC	Add with carry direct
09	17	17	next byte of augend.
0A	97	STA	Store result in
0B	1A	1A	next byte of sum.
0C	96	LDA	Load accumulator direct with
0D	15	15	most significant byte of addend.
0E	99	ADC	Add with carry direct
0F	18	18	most significant byte of augend.
10	97	STA	Store result in
11	1B	1B	most significant byte of sum.
12	3E	HLT	Halt.
13	EA	EA	Least significant byte
14	C0	C0	} Addend.
15	4A	4A	
16	93	93	Least significant byte
17	1B	1B	} Augend
18	66	66	
19	—	—	} Reserved for sum.
1A	—	—	
1B	—	—	

Figure 5-23

Program for multiple-precision addition.

The program for adding the multiple-byte numbers could be written as shown in Figure 5-23. The three byte addend is stored in locations 13<sub>16</sub> through 15<sub>16</sub> while the augend is stored in locations 16<sub>16</sub> through 18<sub>16</sub>. Verify that the hexadecimal contents shown are the same as the binary values given in Figure 5-22.

The first two instructions add the least significant bytes of the addend and augend. The ADD instruction is used because the MPU need not consider earlier carries. The first byte of the resulting sum is stored in location 19<sub>16</sub>.

The next two instructions add the next two bytes. This time the ADC instruction is used because the MPU must consider the carry from the previous addition. The second byte of the sum is placed in location 1A<sub>16</sub>.

Finally, the last two bytes are added using the ADC instruction. The final byte of the sum is stored in location 1B<sub>16</sub>. The program halts when the addition is completed.



## Subtract With Carry (SBC) Instruction

This instruction simplifies multiple-precision subtraction. You will recall that during subtract operations the carry flag indicates whether or not a borrow operation occurred. For this reason, this instruction can be thought of as a subtract with borrow operation.

The SBC instruction subtracts the subtrahend from the minuend just as the SUB instruction did. However, the SBC instruction has an additional step in that the carry bit is also subtracted. As with the other add and subtract instructions, both immediate and direct addressing modes are possible. The opcodes for both modes are shown in Figure 5-21.

Figure 5-24 illustrates how multiple-precision numbers can be subtracted. Notice that, during the course of this subtraction, byte 1 must "borrow" a 1 from byte 2. The SBC instruction allows the MPU to do this.

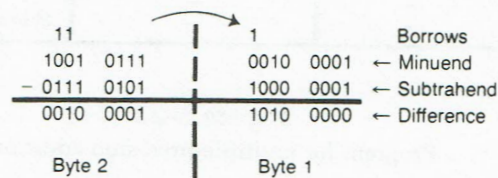


Figure 5-24  
Multiple-precision subtraction.



HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with
01	0D	0D	least significant byte of minuend.
02	90	SUB	Subtract direct
03	0F	0F	least significant byte of subtrahend.
04	97	STA	Store result in
05	11	11	least significant byte of difference.
06	96	LDA	Load accumulator direct with
07	0E	0E	most significant byte of minuend.
08	92	SBC	Subtract with carry
09	10	10	most significant byte of the subtrahend.
0A	97	STA	Store result in
0B	12	12	most significant byte of the difference.
0C	3E	HLT	Halt
0D	21	21	Least significant byte } Minuend.
0E	97	97	Most significant byte }
0F	81	81	Least significant byte } Subtrahend.
10	75	75	Most significant byte }
11	—	—	Least significant byte } Difference.
12	—	—	Most significant byte }

Figure 5-25

Program for multiple-precision subtraction.

Figure 5-25 shows a simple program for performing the subtraction. The double-precision minuend is at addresses  $0D_{16}$  and  $0E_{16}$ , while the subtrahend is at addresses  $0F_{16}$  and  $10_{16}$ . The program computes the difference and stores it in locations  $11_{16}$  and  $12_{16}$ .

The first instruction loads the least significant byte of the minuend. Next, the corresponding byte of the subtrahend is subtracted. Since the subtrahend byte is larger, a borrow is indicated. Consequently, the carry flag is set to 1. Notice that the SUB rather than the SBC instruction is used. This is done because the first byte should not be affected by any previous borrow or carry. The result of the subtraction is stored away to become the least significant byte of the difference.

The most significant byte of the minuend is loaded next and the corresponding byte of the subtrahend is subtracted. However, this time the SBC instruction is used. And since the carry flag is set, an additional 1 is subtracted from the minuend to complete the borrow operation. The result of the subtraction becomes the most significant byte of the difference.

## Arithmetic Shift Accumulator Left (ASLA) Instruction

The ASLA instruction shifts the contents of the accumulator to the left by one bit. Figure 5-26 illustrates the repeated execution of this instruction. Figure 5-26A shows the condition of the accumulator and carry bit. In this example, the number in the accumulator is arbitrarily assumed to be  $10_{10}$ . Also, the carry bit is arbitrarily assumed to be cleared.

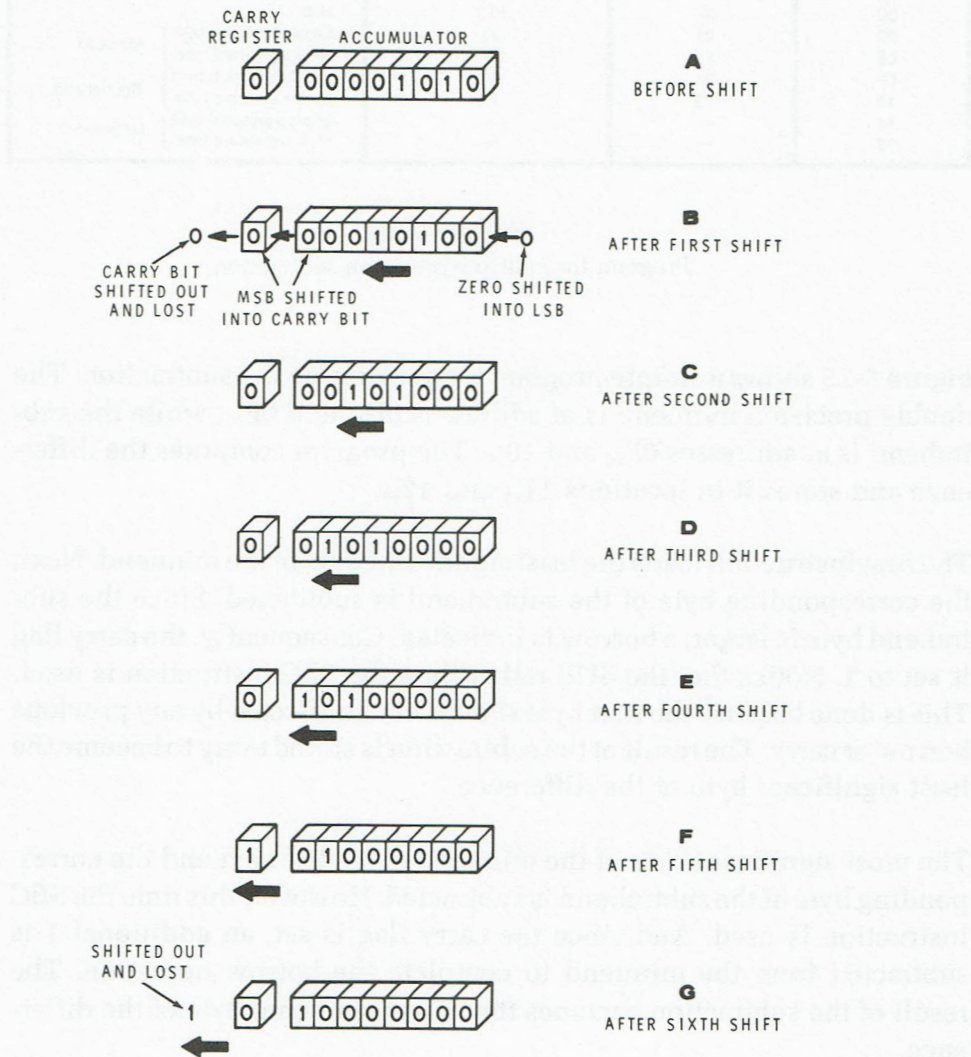


Figure 5-26  
Repeatedly implementing the ASLA instruction.



Figure 5-26B shows the contents of the accumulator and carry bit after the ASLA instruction is executed. Notice that the number is shifted one bit to the left. Also, a 0 is shifted into the LSB. At the same time, the MSB is shifted into the carry bit. The old carry bit is shifted out and is lost.

You can understand one purpose of this instruction by examining the numbers in the accumulator before and after the instruction is executed. Before the shift, the number is  $10_{10}$ ; afterwards the number is  $20_{10}$ . The number has been doubled. If you will try several different examples, you will see that any binary number can be multiplied by two simply by shifting the number one bit to the left. This holds true as long as the capacity of the accumulator is not exceeded.

Figures 5-26C through G show what happens if the MPU continues to execute ASLA instructions. The number continues to double. The number in the accumulator becomes  $40_{10}$ , then  $80_{10}$ , then  $160_{10}$ . Each shift multiplies the number by two. On the fifth shift, the capacity of the accumulator is exceeded as the most significant 1 bit shifts into the carry bit. After the sixth shift, the leading 1 is lost altogether. When you use this technique to multiply by two or by a power of two, you must not exceed the capacity of the accumulator.

Another use of the ASLA instruction is to pack two BCD digits in a single byte. Earlier when we worked with BCD numbers, we assumed that each BCD digit resided in a separate memory byte. However, because a BCD digit has only 4 bits, memory space is wasted by assigning each digit a separate byte. Frequently, it is more desirable to "pack" two BCD digits into a single byte. A simple routine for doing this is shown in Figure 5-27. If dozens of BCD numbers are to be manipulated, a routine that uses a procedure similar to this can save substantial memory space. At the same time, it puts the BCD numbers into a more convenient and usable form.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct
01	0C	0C	the unpacked most significant BCD digit.
02	48	ASLA	} Shift it four places to the left.
03	48	ASLA	
04	48	ASLA	
05	48	ASLA	
06	9B	ADD	Add
07	0D	0D	the unpacked least significant BCD digit.
08	97	STA	Store the result as
09	0B	0B	two packed BCD digits.
0A	3E	HLT	Halt
0B	—	—	Packed BCD digits.
0C	—	—	Most significant BCD digit (unpacked).
0D	—	—	Least significant BCD digit (unpacked).

Figure 5-27

Program for "packing" two BCD digits into a single byte.



The procedure carried out by the program is quite simple. The most significant BCD digit is loaded into the accumulator. It is then shifted four places to the left to make room for the least significant BCD digit. The least significant digit is then added to form a packed BCD number. The resulting single byte number is stored back in memory.

## Decimal Adjust Accumulator (DAA) Instruction

Earlier in this unit, the problems of converting from BCD to binary and back again were considered. While this conversion is frequently necessary, many microprocessors have some limited BCD arithmetic capabilities. Our hypothetical MPU has an instruction that greatly simplifies BCD arithmetic. It is called the Decimal Adjust Accumulator (DAA) Instruction. When used in conjunction with the ADD or ADC instruction, it allows the MPU to add BCD numbers directly without an intermediate binary conversion.

Recall that the ALU adds input data bytes as if they were unsigned binary numbers. Therefore, if two BCD digits are added, the sum may be incorrect. For example, assume that the MPU adds the BCD digits 0111 and 0101. The ALU produces the result

$$\begin{array}{r} 1 \\ 0111 \\ + 0101 \\ \hline 1100 \end{array}$$

This answer is the correct **binary** result,  $12_{10}$ ; but it is not the proper BCD result. Recall that in BCD,  $12_{10}$  is represented as 0001 0010. Notice that you can obtain the proper BCD result by adding  $0110_2$  to the binary result. The addition of  $0110_2$  is necessary anytime that the binary result exceeds  $1001_2$ .

To produce the proper BCD result when adding two BCD digits, the MPU must follow this procedure:

1. If the sum is  $1001_2$  or less, use the sum as the single digit BCD result.
2. If the sum is greater than  $1001_2$ , add  $0110_2$  and use the result as a 2-digit BCD number.

$$\begin{array}{r} 11 \qquad 1 \\ 0111 \quad 1001 \\ 0111 \quad 0011 \\ \hline 1110 \quad 1100 \end{array}$$
$$\begin{array}{r} 1 \quad 1 \quad 1 \quad 1 \\ \phantom{1} 1110 \quad 1100 \\ \phantom{1} 0110 \quad 0110 \\ \hline 1 \quad 0101 \quad 0010 \end{array}$$

If you consider all possible combinations of BCD numbers, you will find that four different situations exist:

1. When some BCD numbers are added, the binary result pro-

1. When some BCD numbers are added, the binary result produced by the ALU is equal to the proper BCD representation. This occurs when both BCD digits of the result are  $1001_2$  or less.

2. If the least significant BCD digit exceeds  $1001_2$  but the most significant BCD digit does not, the binary sum is adjusted by adding  $06_{16}$ .
3. If the most significant BCD digit exceeds  $1001_2$  but the least significant digit does not, the binary sum is adjusted by adding  $60_{16}$ .
4. If both BCD digits exceed  $1001_2$ , the binary sum is adjusted by adding  $66_{16}$ .



While this procedure could be programmed, it would be much better if the MPU performed these operations automatically. Fortunately, our hypothetical microprocessor does this. The programmer simply informs the MPU that the numbers being added are BCD numbers. The MPU automatically computes the proper BCD result. The way the programmer informs the MPU is via the DAA instruction. When the DAA instruction is placed immediately after an ADD or ADC instruction, the MPU automatically converts the sum to the proper BCD number.

Suppose, for example, that you wish to add two BCD numbers. Assume the numbers are  $3792_{10}$  and  $5482_{10}$ . Naturally, the sum should be  $9274_{10}$ . A program for solving this problem is shown in Figure 5-28. The BCD addend ( $3792_{10}$ ) is in addresses  $0F_{16}$  and  $10_{16}$ . The augend ( $5482_{10}$ ) is in locations  $11_{16}$  and  $12_{16}$ . The BCD sum will be placed in locations  $13_{16}$  and  $14_{16}$ .

The first two instructions add the least significant halves of the addend and augend. The ADD instruction is followed immediately by the DAA instruction. Therefore, the sum is adjusted to a packed BCD number. The result is stored in location  $14_{16}$  as the lower half of the BCD sum.

Next, the upper halves of the addend and augend are added. This time, the ADC instruction is used because the carry from the previous addition must be added in. Again, the DAA instruction adjusts the sum to BCD. The result is stored as the upper half of the BCD sum.

The DAA instruction must be used properly. It can be used only with addition. Also, it must be used immediately after the addition instruction. It can not be used to convert just any binary number to BCD. It only converts the sum of BCD numbers to the BCD format.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct
01	10	10	the least significant half of the addend.
02	9B	ADD	Add
03	12	12	the least significant half of the augend.
04	19	DAA	Decimal adjust the sum to BCD.
05	97	STA	Store the result as the
06	14	14	least significant half of the sum.
07	96	LDA	Load
08	0F	0F	the most significant half of the addend.
09	99	ADC	Add with carry.
0A	11	11	the most significant half of the augend.
0B	19	DAA	Decimal adjust the sum to BCD.
0C	97	STA	Store the result as the
0D	13	13	most significant half of the sum.
0E	3E	HLT	Halt
0F	37	37	} BCD Addend
10	92	92	
11	54	54	} BCD Augend.
12	82	82	
13	—	—	} Reserved for BCD sum.
14	—	—	

Figure 5-28

Program for adding multiple-precision BCD numbers.



## Programmed Review

28. The ALU always adds numbers as if they were \_\_\_\_\_  
binary numbers. (signed/unsigned)

29. (unsigned) If the C flag is set to 1 before the ADC instruction is executed, 1 is added to the \_\_\_\_\_ significant bit of the sum. (most/least)

30. (least) The ADC instruction has \_\_\_\_\_ addressing mode(s).

31. (two) A primary use of the ADC and SBC instructions is in \_\_\_\_\_ - \_\_\_\_\_ arithmetic.

32. (multiple-precision) In an 8-bit microprocessor, multiple-precision means that two or more \_\_\_\_\_ are used to represent a number.

33. (bytes) The ASLA instruction shifts the contents of the \_\_\_\_\_ to the left by one bit.

34. (accumulator) Using the ASLA instruction, any binary number can be \_\_\_\_\_ by two by shifting the number one bit to the left. (multiplied/divided)

35. (multiplied) The accumulator contains the number  $7_{10}$ . If three ASLA instructions are executed, the number \_\_\_\_\_ will be in the accumulator.

36. ( $56_{10}$ ) When using the ASLA instruction to multiply by two or by a power of two, it \_\_\_\_\_ possible to exceed the capacity of the accumulator.  
(is/is not)

37. (is) With \_\_\_\_\_ BCD numbers, each byte contains one BCD digit.  
(packed/unpacked)

38. (unpacked) The \_\_\_\_\_ instruction, when used in conjunction with the ADD or ADC instruction, allows the MPU to add BCD numbers directly without an intermediate binary conversion.

39. (Decimal Adjust Accumulator (DAA)) The DAA instruction \_\_\_\_\_ be used with subtraction.  
(can/cannot)

40. (cannot) The DAA instruction is used immediately \_\_\_\_\_ the addition instruction.  
(before/after)

(after)

## EXPERIMENTS

Perform Programming Experiments 7 and 8. You will find these experiments in Unit 12. After you finish these experiments, return to this unit and complete the Unit Examination.



## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct ones that appear after the exam.

1. The BRA instruction will cause a branch to occur:
  - A. Anytime that it is executed.
  - B. Only if the Z flag is set.
  - C. Only if the N flag is set.
  - D. Only if the C flag is set.
2. The address that follows the opcode of an unconditional branch instruction is:
  - A. The address of the operand.
  - B. The address of the next opcode to be executed.
  - C. Added to the program count to form the address of the next opcode to be executed.
  - D. Added to the program count to form the address of the operand that is to be tested to see if a branch operation is required.
3. The opcode for an unconditional branch instruction is at address  $AF_{16}$ . The relative address is  $0F_{16}$ . From what address will the next opcode be fetched?
  - A.  $A0_{16}$ .
  - B.  $C0_{16}$ .
  - C.  $BE_{16}$ .
  - D.  $B1_{16}$ .

4. The opcode for an unconditional branch instruction is at address  $30_{16}$ . The relative address is  $EF_{16}$ . From what address will the next opcode be fetched?
- A.  $21_{16}$ .
  - B.  $EF_{16}$ .
  - C.  $32_{16}$ .
  - D.  $19_{16}$ .
5. The carry register:
- A. Acts like the ninth bit of the accumulator.
  - B. Is set when a "borrow" for bit 7 of the accumulator occurs.
  - C. Is set when a carry from bit 7 occurs.
  - D. All of the above.
6. The number  $0101\ 1000_2$  and  $0110\ 0011_2$  are added using the ADD instruction. Immediately after the ADD instruction is executed, the condition code registers will indicate the following:
- A.  $C=1, N=1, V=1, Z=0$ .
  - B.  $C=0, N=1, V=1, Z=0$ .
  - C.  $C=0, N=1, V=0, Z=0$ .
  - D.  $C=0, N=0, V=1, Z=1$ .
7. The divide program shown in Figure 5-16 works only if the dividend is initially less than  $+128_{10}$ . The program can be modified to work for dividends up to  $255_{10}$  by replacing the BMI instruction with the:
- A. BEQ instruction.
  - B. BNE instruction.
  - C. BCC instruction.
  - D. BCS instruction.



8. A binary number can be converted to BCD by repeatedly:
  - A. Dividing by powers of two.
  - B. Subtracting powers of ten.
  - C. Multiplying by powers of two.
  - D. Adding powers of ten.
  
9. The DAA instruction is used:
  - A. To convert a binary number to BCD.
  - B. To convert a BCD number to binary.
  - C. After an add instruction to adjust the sum to a BCD number.
  - D. After a subtract instruction to adjust the difference to a BCD number.
  
10. When you are adding multiple-precision binary numbers, all bytes except the least significant ones must be:
  - A. Added using the ADD instruction.
  - B. Added using the DAA instruction.
  - C. Added using the ADC instruction.
  - D. Decimal adjusted before addition takes place.

## EXAMINATION ANSWERS

For your convenience, the page(s) where the correct answer can be found is shown following the answer.

1. A —Anytime that it is executed. [19]
2. C —Added to the program count to form the address of the next opcode to be executed. [13]
3. B — $C0_{16}$ . [13]
4. A — $21_{16}$ . [16]
5. D —All of the above. [22]
6. B — $C=0, N=1, V=1, Z=0$ . [26]
7. D —BCS instruction. [23]
8. B —Subtracting powers of ten. [41]
9. C —After an add instruction to adjust the sum to a BCD number. [54]
10. C —Added using the ADC instruction. [46]

## *Unit 6*

# **A TYPICAL MICROPROCESSOR CONTROLLER**

## CONTENTS

Introduction .....	6-3
Unit Objectives .....	6-4
Unit Activity Guide .....	6-5
Architecture of the 6808 MPU .....	6-6
Instruction Set of the 6808 MPU .....	6-16
New Addressing Modes .....	6-39
Experiments .....	6-52
Stack Operations .....	6-54
Subroutines .....	6-66
Input/Output (I/O) Operations .....	6-76
Interrupts .....	6-87
Experiments .....	6-104
Unit Examination .....	6-105
Examination Answers .....	6-111

## INTRODUCTION

Until now, we have confined our study to a hypothetical microprocessor. Obviously, though, this hypothetical model must be very close to the real thing, since we have run its programs on the ET-18 Robot Trainer. Our hypothetical model was based on the 6800 microprocessor family. In this unit, you will study an actual microprocessor, the 6808, which in reality could be used to control an industrial robot.

You already know a great deal about the 6808 microprocessor since your Robot Trainer contains one. The main difference between the 6808 microprocessor and our hypothetical one is complexity. As you will see, the 6808 is a vastly expanded version of our hypothetical model.

The 6808 is part of a family that began with the 6800 and has developed over the past few years to include several other devices. The 6808 uses the basic 6800 instruction set which you are learning in this course. Some of the devices, however, have additional instructions and addressing modes, on-board memory, and facilities for direct connection to input/output devices. Features such as these make one microprocessor more powerful, versatile, or more suitable for a particular application than another. For your reference, the data sheets in Appendix B give complete specifications for the 6808 microprocessor.

This unit is basically divided into two parts. In the first part, you will be introduced to the architecture and instruction set of the 6808 microprocessor. Also, much of the MPU's capabilities will also be discussed. In the second section, three very important microprocessor operations — stack operations, subroutines, and interrupt capabilities — will be discussed in detail.



## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Explain the function of each block in a simplified block diagram of the 6808 MPU.
2. Using Appendix C as a reference, explain the operation of all the instructions discussed in this unit.
3. Write simple programs that use indexed and extended addressing.
4. Using Figure 6-24 as a guide, find the opcode, number of MPU cycles, number of bytes, and effects on the condition code flags of every instruction discussed in this unit.
5. Explain the difference between a cascade stack and a memory stack.
6. Write simple programs that can store data in—and retrieve data from—the stack.
7. Write programs that use the stack and indexing to move a list from one place in memory to another.
8. Explain the operations performed by each of the following instructions: PULA, PULB, PSHA, PSHB, DES, INS, LDS, STS, TXS, and TSX.
9. Define stack, subroutine, nested subroutine, interrupt, interrupt vector, and interrupt masking.
10. Explain the operations performed by each of the following instructions: JMP, JSR, BSR, and RTS.
11. Describe how the 6808 MPU performs input and output operations.
12. Explain the operation performed by each of the following instructions: WAI, SWI, RTI, SEI, and CLI.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "Architecture Of The 6808 MPU."	_____
<input type="checkbox"/> Answer Programmed Review Questions 1-11.	_____
<input type="checkbox"/> Read "Instruction Set Of The 6808 MPU."	_____
<input type="checkbox"/> Answer Programmed Review Questions 12-25.	_____
<input type="checkbox"/> Read "New Addressing Modes."	_____
<input type="checkbox"/> Perform Programming Experiments 9 and 10.	_____
<input type="checkbox"/> Answer Programmed Review Questions 26-34.	_____
<input type="checkbox"/> Read "Stack Operations."	_____
<input type="checkbox"/> Answer Programmed Review Questions 35-41.	_____
<input type="checkbox"/> Read "Subroutines."	_____
<input type="checkbox"/> Answer Programmed Review Questions 42-50.	_____
<input type="checkbox"/> Read "Input/Output (I/O) Operations."	_____
<input type="checkbox"/> Answer Programmed Review Questions 51-56.	_____
<input type="checkbox"/> Read "Interrupts."	_____
<input type="checkbox"/> Answer Programmed Review Questions 57-67.	_____
<input type="checkbox"/> Perform Programming Experiments 11 and 12.	_____
<input type="checkbox"/> Complete The Unit Examination.	_____
<input type="checkbox"/> Check The Examination Answers.	_____

## ARCHITECTURE OF THE 6808 MPU

In computer terminology, the word architecture is used to describe the computer's style of construction: its register size and arrangement, its bus configuration, etc. The architecture of our hypothetical microprocessor is shown for one last time in Figure 6-1. By now, you should be quite familiar and comfortable with this architecture.

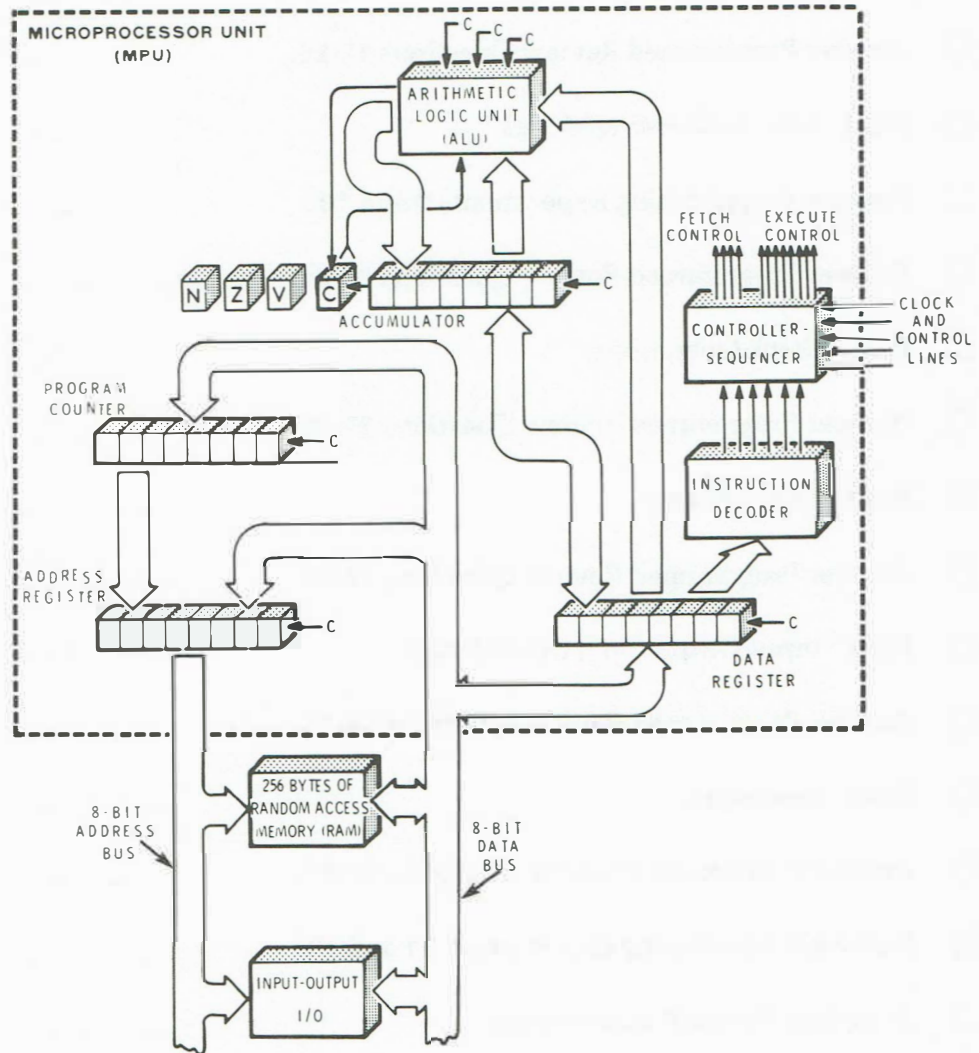


Figure 6-1

Architecture of the hypothetical microcomputer.

The only reason for showing you the details of the model is to give you an idea of what goes on inside the integrated circuit. In an actual microprocessor, the internal structure is often so complex that we become bogged down in the details if we attempt to analyze it too closely. For this reason, a programming model is generally used when a microprocessor is being introduced for the first time. In the programming model, any register or circuit that cannot be controlled by the program is simply ignored. Consider the data register for example. There are no instructions that give the program direct control over this register. That is, there are no instructions such as Load Data Register, Store Data Register, etc. All data register activity is controlled strictly by the MPU. Thus, the program can simply ignore the existence of the register.

The same is true of the address register, the instruction decoder, the controller-sequencer, etc. Therefore, the programming model of our hypothetical MPU can be represented as shown in Figure 6-2. This simple diagram is sufficient for most programming applications, since it shows all the registers that can be directly controlled by the program.

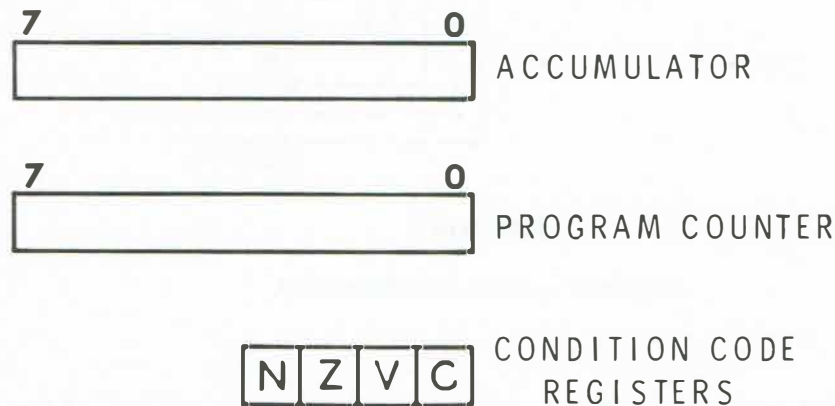


Figure 6-2

Programming model of the hypothetical MPU.

## Programming Model of the 6808 MPU

The 6808 MPU is much more complex than our hypothetical MPU. Consequently, a programming model of the 6808 makes a good starting point. The programming model is shown in Figure 6-3.

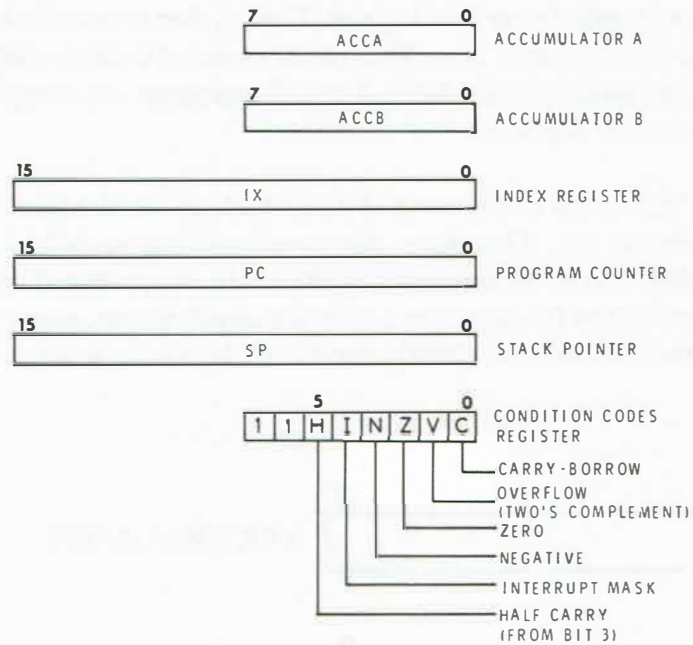


Figure 6-3

Programming model of the 6808 MPU.

You will notice immediately that the 6808 MPU has several additional registers. However, only two of these, the index register and the stack pointer, are actually new to you. Let's look at the major differences between this MPU and our hypothetical model.



**Two Accumulators**—The 6808 MPU has two accumulators instead of one. They are called accumulator A and accumulator B, and are referred to as (ACCA) and (ACCB) respectively. Each has its own group of instructions associated with it. The names and mnemonics of the instructions specify which accumulator is to be used. Thus, there are instructions such as:

Load Accumulator A (LDAA)  
Load Accumulator B (LDAB)  
Store Accumulator A (STAA)  
Store Accumulator B (STAB)

Notice that a letter is added to both the name and the mnemonic to indicate which accumulator is being used.

From your previous programming experience, you can visualize the value of a second accumulator. For example, consider a program in which the MPU counts the number of times that some operation occurs. In the past, we stored the number that the accumulator was presently working on, loaded the count into the accumulator, incremented the count, stored the count, and reloaded the original number. With a second accumulator, none of this is necessary. We can simply maintain the count in accumulator B while working with the number in accumulator A. In fact, we can perform any arithmetic or logic operation on two different numbers without having to shift the numbers back and forth between memory.

**16-Bit Program Counter**—The program counter in the 6808 has  $16_{10}$  bits rather than  $8_{10}$ . Thus, it can specify  $65,536_{10}$  different addresses. This means that a 6808 based microcomputer can have up to  $65,536_{10}$  bytes of memory. Most applications require substantially less memory than this maximum number. Fortunately, we can use as little or as much memory as we need, up to the  $2^{16}$  byte limit.

Since the program counter has  $16_{10}$  bits, the address bus must also be 16-bits wide. Contrast this with the 8-bit address of our hypothetical machine.

You may wonder how we specify a 16-bit address with an 8-bit byte. The answer is that two 8-bit bytes are required. Recall that in the direct addressing mode, the address was specified by a single 8-bit byte. The 6808 microprocessor retains this addressing mode. However, since an 8-bit address can specify only  $256_{10}$  addresses, the 6808 MPU can use this mode only if the operand is in the first  $256_{10}$  bytes of memory. To reach higher addresses, a new address mode called **extended addressing** must be used. In the extended addressing mode, two bytes are used to represent each address. This addressing mode will be discussed in more detail later. For now, keep in mind that there are  $65,536_{10}$  possible addresses. The lowest address is  $0000_{16}$  and the highest is  $FFFF_{16}$ . Using extended addressing, we have access to any location in memory, but a 2-byte address is required.

**Condition Code Registers**—The 6808 MPU has six condition codes. Four of these are almost identical to those discussed in Unit Five. As you recall, these include the negative (N), zero (Z), overflow (V) and carry (C) condition codes. One difference arises, however, because there are two accumulators in the 6808 MPU. Thus, the carry (C) flag is set whenever there is a carry from either accumulator. By the same token, an overflow in either accumulator will set the overflow (V) flag. Later in this unit, you will see how the condition codes are affected by each other.

Two new condition codes are shown in Figure 6-3. The (I) flag is called an **interrupt mask**. We will discuss this flag later when you study interrupts. The other is called the **half-carry** (H) flag. The (H) flag is set when there is a carry from bit 3 of the accumulator. The MPU uses this flag to determine how to implement the decimal adjust instruction.

These six flags make up bits 0 through 5 of an 8-bit register. Bits 6 and 7 of the condition code register are not used and are always set to 1. Additional details of the condition codes will be brought out as the need arises.

**Index Register**—The index register is a special purpose, 16-bit register that greatly increases the power of the microprocessor. It allows a powerful address mode called **indexed addressing** to be used. We will examine this addressing mode later in the unit. For now, consider the index register to be just another working register. The fact that it holds two bytes instead of one can be put to good use. The MPU has instructions that allow the index register to be loaded from two adjacent memory bytes. Another instruction allows us to store the contents of the index register in two adjacent memory locations. This allows us to move data in 2-byte groups. Also, the index register can be incremented and decremented, which permits us to maintain 16-bit tallies.

**Stack Pointer**—The stack pointer is another special-purpose 16-bit register. It allows the MPU and the programmer to use a section of RAM as a last-in/first-out (LIFO) memory. This capability is extremely valuable when using subroutines or when processing interrupts. These aspects of the stack pointer will be discussed in greater detail later in this unit. For the time being, let's consider the stack pointer to be another 16-bit working register. Its contents also can be loaded from memory, stored in memory, incremented, and decremented.

## Block Diagram of the 6808 MPU

Now that you have seen the programming model of the 6808 MPU, take a look at the simplified block diagram of the 6808 MPU shown in Figure 6-4. Several data paths, most control lines, and a temporary storage register have been omitted in favor of the major data paths and registers.

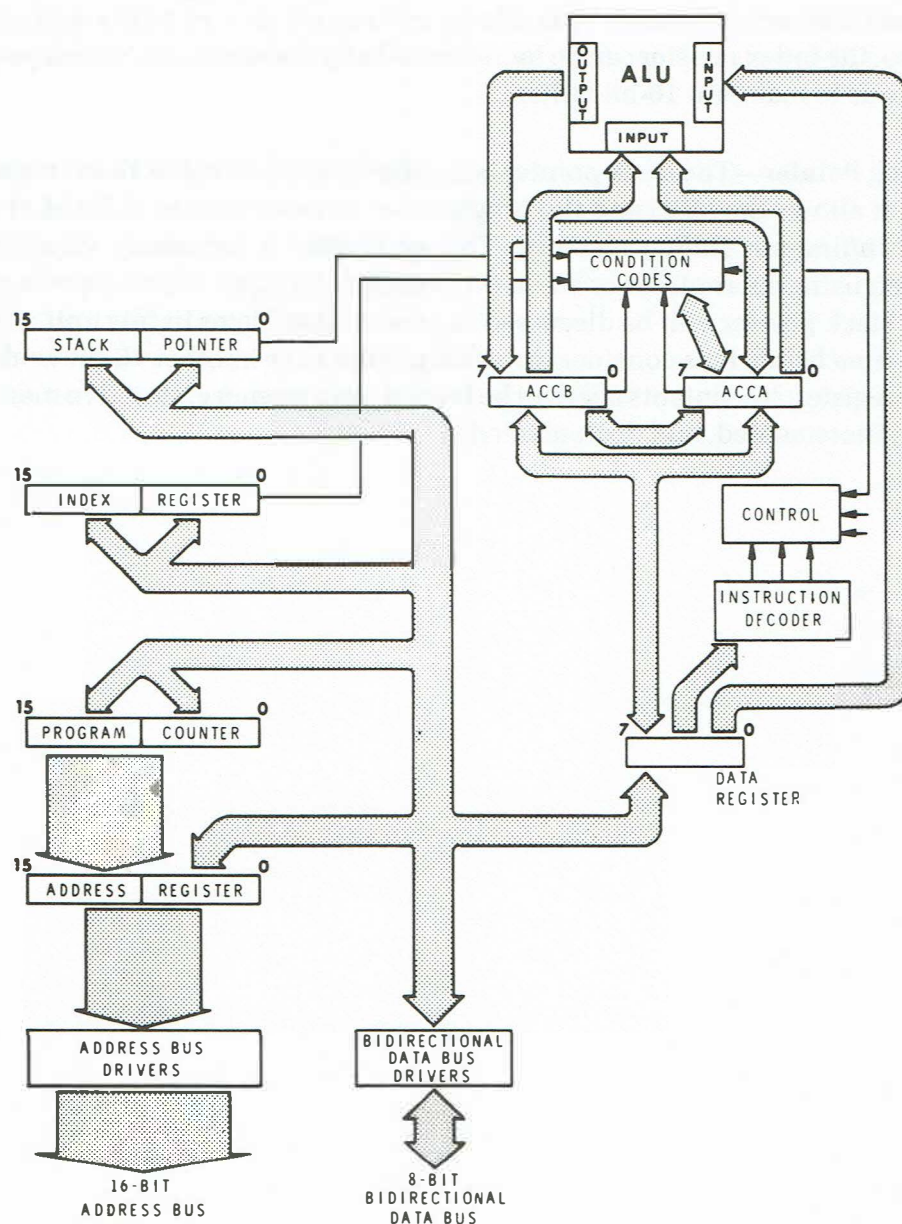


Figure 6-4

Simplified block diagram of the 6808 MPU.

The 16-bit registers shown on the left are primarily concerned with addressing memory. Since the address bus has 16-bits, all registers can be loaded from the data bus. However, because the data bus has only eight bits, two operations are required to load the 16-bit registers. The upper half of the affected register is always loaded first. Then, a second operation loads the lower half. Although this requires separate MPU cycles, the microprocessor takes care of these operations automatically. For example, a single instruction can load the 16-bit index register with two memory bytes.

The program counter and address register perform exactly the same functions in the 6808 MPU as they did in our hypothetical model. The fetch and execute phases for the immediate and direct addressing modes are virtually identical. The same is true of the relative addressing mode except that the 8-bit relative address is added to the 16-bit program count.

The 8-bit registers are shown on the right. Notice that these circuits are identical to those in our hypothetical model except that there are two accumulators. The condition code registers monitor both accumulators. Also, the two accumulators share the ALU. This allows you to keep track of two separate mathematical operations at more or less the same time. This arrangement is particularly flexible since the contents of one accumulator can be transferred to the other, or the contents of the two accumulators can be added together.



## Programmed Review

1.	The microprocessor on which our hypothetical model and the ET-18 Robot Trainer are based is the _____ MPU.
2.	(6808) A major difference between our hypothetical model and the 6808 MPU is that the 6808 has two _____.
3.	(accumulators) The program counter in the 6808 MPU has _____ bits.
4.	( $16_{10}$ ) The 6808-based microcomputer has an address bus that is _____ wide.
5.	( $16_{10}$ bits) The addresses in the 6808 MPU range from $0000_{16}$ to _____.
6.	( $FFFF_{16}$ ) In the 6808 MPU, the _____ flag is set when there is a carry from bit 3 of the accumulator.
7.	(half-carry (H)) The 6808 MPU uses the (H) flag to determine how to implement the _____ instruction.
8.	(decimal adjust) The six condition code flags used with the 6808 MPU make up bits _____ through _____ of an 8-bit register.

9.	(0, 5) The index register is a special-purpose, _____-bit register that greatly increases the power of the microprocessor.
10.	(16) In the 6808 MPU, the two accumulators _____ share a common carry flag. (do/do not)
11.	(do) In the 6808 MPU, the contents of the two accumulators _____ be added together. (can/cannot)
	(can)

## INSTRUCTION SET OF THE 6808 MPU

The 6808 MPU has about 100 basic instructions. Moreover, when all the different addressing modes are considered, there are 197 different opcodes to which the MPU will respond.

These instructions can be broken down into seven general categories. While some of these categories overlap, the general classifications of instructions are:

1. arithmetic.
2. data handling.
3. logic.
4. data test.
5. index register and stack pointer.
6. jump and branch.
7. condition code.

In this discussion, we will not be concerned with addressing modes. Therefore, no opcodes are given. Later, we will look at the various addressing modes and opcodes. For now, we will identify the instructions by their names, mnemonics, and operations. You will see what each instruction does and how it affects the various condition code registers.

Because of the large number of instructions covered in this section, the explanations will be general and brief. You are not expected to remember all of the details of every instruction. “**Appendix C**” of this course contains a detailed listing of each instruction. It explains every detail of the various instructions. After reading this section, turn to “Appendix C” and look over the explanations given there. In the future, when you are in doubt as to exactly what a particular instruction does, look it up in “**Appendix C.**”

## Arithmetic Instructions

Figure 6-5 shows the arithmetic instructions of the 6808 MPU. The name of each instruction is given on the left. The next column contains the mnemonics. The center column gives a shorthand description of what the instruction does. The right-hand columns show how the various condition code registers are affected.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Add	ADDA	$A + M \rightarrow A$	↑	●	↑	↑	↑	↑
	ADDB	$B + M \rightarrow B$	↑	●	↑	↑	↑	↑
Add Acmltrs	ABA	$A + B \rightarrow A$	↑	●	↑	↑	↑	↑
Add with Carry	ADCA	$A + M + C \rightarrow A$	↑	●	↑	↑	↑	↑
	ADCB	$B + M + C \rightarrow B$	↑	●	↑	↑	↑	↑
Complement, 2's (Negate)	NEG	$00 - M \rightarrow M$	●	●	↑	↑	①	②
	NEGA	$00 - A \rightarrow A$	●	●	↑	↑	①	②
	NEGB	$00 - B \rightarrow B$	●	●	↑	↑	①	②
Decimal Adjust, A	DAA	Converts Binary Add. of BCD Characters into BCD Format*	●	●	↑	↑	↑	③
Subtract	SUBA	$A - M \rightarrow A$	●	●	↑	↑	↑	↑
	SUBB	$B - M \rightarrow B$	●	●	↑	↑	↑	↑
Subtract Acmltrs.	SBA	$A - B \rightarrow A$	●	●	↑	↑	↑	↑
Subtr. with Carry	SBCA	$A - M - C \rightarrow A$	●	●	↑	↑	↑	↑
	SBCB	$B - M - C \rightarrow B$	●	●	↑	↑	↑	↑

\*Used after ABA, ADC, and ADD in BCD arithmetic operation; each 8-bit byte regarded as containing two 4-bit BCD numbers. DAA adds 0110 to lower half-byte if least significant number >1001 or if preceding instruction caused a Half-carry. Adds 0110 to upper half-byte if most significant number >1001 or if preceding instruction caused a Carry. Also adds 0110 to upper half-byte if least significant number >1001 and most significant number = 9.

(Bit set if test is true and cleared otherwise)

- ① (Bit V) Test: Result = 10000000?
- ② (Bit C) Test: Result ≠ 00000000?
- ③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine?  
(Not cleared if previously set.)

Figure 6-5

Arithmetic instructions.

To reinforce your understanding of this idea, we will go through the first instruction in detail. The first instruction is the add instruction. Actually, since the 6808 has two accumulators, there are two add instructions. Their mnemonics are ADDA and ADDB. Notice that the final letter of the mnemonic indicates whether accumulator A or B is involved. The shorthand representation of the operation is:  $A + M \rightarrow A$ . The note at the top of this column tells you that the register labels refer to the contents of the register. Thus, A means the contents of accumulator A and M means the contents of the affected memory location. The symbol ( $\rightarrow$ ) means "Transfer Into." Therefore,  $A + M \rightarrow A$  means "Add the contents of accumulator A to the contents of the affected memory location and transfer the sum into accumulator A."

To see how the condition code flags are affected, you simply look over to the right under whatever condition code you are interested in. Generally, the condition code is either unaffected or is tested and set accordingly. When the condition code is unaffected, this is represented by the symbol (●). For example, none of the arithmetic instructions affect the (I) flag. Most of the arithmetic instructions test the condition codes and set them if the condition exists. For example, if the result of an arithmetic operation is zero, the (Z) flag is set to 1. Consequently, if this condition does not exist, the (Z) flag is reset or cleared to 0. The symbol ( $\updownarrow$ ) means "test and set if true; clear otherwise." Occasionally, a note is necessary to describe some unusual situation regarding the condition code. This is represented by a number within a circle. An example of these notes is given at the bottom of the drawing in Figure 6-5.

The ADDA and ADDB instructions are self-explanatory. The ABA instruction adds the contents of accumulator A to the contents of accumulator B. The result is stored in accumulator A.

The add-with-carry instructions are identical to those discussed earlier for our hypothetical machine. Notice that the carry bit is added in with the sum.

Because two's complement arithmetic is used in the 6808 MPU, instructions are provided that allow us to take the two's complement of a number. The negate instruction subtracts the contents of the affected register from  $00_{16}$ . This is the same as taking the two's complement of the number. The affected register can be any memory location (M) or either accumulator (A or B). Thus, there are three different negate instructions. Keep in mind that NEG means "take the two's complement of the affected memory location;" NEGA means "take the two's complement of accumulator A;" and NEGB means "take the two's complement of accumulator B."



Notice that the NEG instruction allows us to operate on a byte of memory without first fetching the operand from memory. In the past, we have loaded the operand, performed the operation, and then stored the new operand. However, the 6808 allows us to perform certain operations on the operand without first fetching it from memory. Several examples of this will be pointed out as we progress through the instruction set.

The decimal adjust instruction performs exactly as it did in our hypothetical machine. The note immediately under the table summarizes its operation. However, this instruction works only with accumulator A.

The subtract and subtract-with-carry instructions are self-explanatory. They perform as described earlier for our hypothetical MPU. The 6808 MPU has an additional subtract instruction. The SBA instruction subtracts the contents of accumulator B from the contents of accumulator A, with the resulting difference being placed in accumulator A.

## Data-Handling Instructions

Figure 6-6 shows the largest group of instructions used by the 6808 MPU. These can be loosely categorized as data-handling instructions.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Clear	CLR	00 → M	●	●	R	S	R	R
	CLRA	00 → A	●	●	R	S	R	R
	CLRB	00 → B	●	●	R	S	R	R
Decrement	DEC	M - 1 → M	●	●	↑	↑	④	●
	DECA	A - 1 → A	●	●	↑	↑	④	●
	DECB	B - 1 → B	●	●	↑	↑	④	●
Increment	INC	M + 1 → M	●	●	↑	↑	⑤	●
	INCA	A + 1 → A	●	●	↑	↑	⑤	●
	INCB	B + 1 → B	●	●	↑	↑	⑤	●
Load Acmltr	LDAA	M → A	●	●	↑	↑	R	●
	LDAB	M → B	●	●	↑	↑	R	●
Rotate Left	ROL	M	●	●	↑	↑	⑥	↑
	ROLA	A	●	●	↑	↑	⑥	↑
	ROLB	B	●	●	↑	↑	⑥	↑
Rotate Right	ROR	M	●	●	↑	↑	⑥	↑
	RORA	A	●	●	↑	↑	⑥	↑
	RORB	B	●	●	↑	↑	⑥	↑
Shift Left, Arithmetic	ASL	M	●	●	↑	↑	⑥	↑
	ASLA	A	●	●	↑	↑	⑥	↑
	ASLB	B	●	●	↑	↑	⑥	↑
Shift Right, Arithmetic	ASR	M	●	●	↑	↑	⑥	↑
	ASRA	A	●	●	↑	↑	⑥	↑
	ASRB	B	●	●	↑	↑	⑥	↑
Shift Right, Logic	LSR	M	●	●	R	↑	⑥	↑
	LSRA	A	●	●	R	↑	⑥	↑
	LSRB	B	●	●	R	↑	⑥	↑
Store Acmltr	STAA	A → M	●	●	↑	↑	R	●
	STAB	B → M	●	●	↑	↑	R	●
Transfer Acmltrs	TAB	A → B	●	●	↑	↑	R	●
	TBA	B → A	●	●	↑	↑	R	●

- ④ (Bit V) Test: Operand = 10000000 prior to execution?  
 ⑤ (Bit V) Test: Operand = 01111111 prior to execution?  
 ⑥ (Bit V) Test: Set equal to result of  $N \oplus C$  after shift has occurred.

Figure 6-6

Data handling instructions.

The clear instructions allow us to clear a memory location or either accumulator. Previously, we had to clear bytes of memory by first clearing the accumulator and then storing the resulting 00 in the proper memory location. However, the CLR instruction allows us to clear a memory location with a single instruction. Notice that some new entries appear in the condition code register column. (R) means that the condition code is always reset or cleared to 0. (S) means that the code is always set to 1.

The decrement instruction allows us to subtract 1 from a memory location or from either accumulator. The DEC instruction is especially valuable, since it allows us to decrement to a byte in memory with a single instruction. Previously, we loaded the byte, decremented it, and then stored it back in memory.

The increment instructions are similar except that they allow us to add 1 to a memory location or to one of the accumulators. Notice that the INC instruction allows us to maintain a tally in memory without having to load it, increment it, and then store it away.

The load accumulator instructions are self-explanatory. Notice that either accumulator can be loaded from memory.

The rotate-left instructions allow us to shift the contents of the accumulator or a memory location without losing bits of data. Consider the ROLA instruction as an example. When this instruction is executed, the A accumulator and the carry bit form a 9-bit circulating register. That is, they form a closed loop as shown in Figure 6-7A. When ROLA is executed, the data is rotated clockwise. The MSB of accumulator A shifts into the carry register. Simultaneously, the contents of accumulator A are shifted left. Notice that the carry bit is not lost; instead, it is shifted into the LSB of the accumulator.

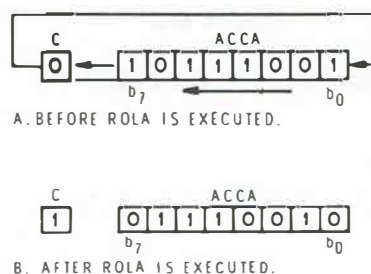


Figure 6-7

Executing the ROLA instruction.

While the usefulness of this instruction may not be obvious, it is a valuable tool. For instance, it can be used to determine parity. Parity is a summation check in which the binary digits, in a character or word, are added, and the sum checked against a previously computed parity digit. That is, a check for parity determines whether the number of 1's in a word is odd or even. By repeatedly rotating left and testing the (C) flag, you can determine the number of 1's in the byte. Once you know this, you could easily generate the proper parity bit.

The ROL instruction allows you to rotate a memory byte to the left while it is still in memory. ROLB allows you to rotate the B accumulator to the left. In each case, the (C) register is used as a ninth bit.

The rotate-right instructions are identical except that the direction of rotation is reversed. Figure 6-8 illustrates the execution of the RORA instruction. This instruction is also valuable. Suppose, for example, that we wish to know if the number in the accumulator is even or odd. This is determined by the LSB of the number. If the LSB = 1, the number is odd; if the LSB = 0, the number is even. One way to determine this is to rotate the number to the right so that the LSB is in the (C) register. We could then test the (C) register to see if it is set or cleared. Notice that the number could then be restored to its original value by the ROLA instruction.

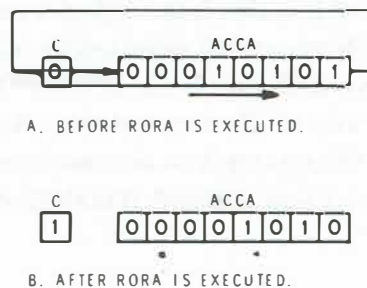


Figure 6-8

Executing the RORA instruction.

The arithmetic shift-left instruction was discussed earlier in our hypothetical MPU. The ASLA instruction performs exactly as described in the previous unit. However, notice that the 6808 MPU also has an ASLB instruction that performs the same operation with accumulator B. Also, it has an ASL instruction that allows us to perform this operation on a byte that is in memory. Figure 6-9 illustrates the execution of this instruction.

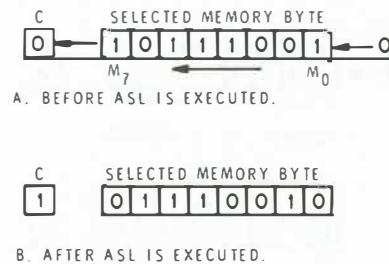


Figure 6-9

Executing the ASL instruction.

While there is only one type of shift-left instruction, there are two types of shift-right instructions: arithmetic shift-right and logic shift-right instructions. We will discuss the arithmetic shift-right instruction first.

When an arithmetic shift-right instruction is executed, the number in the affected register is shifted right one position, and the LSB goes into the (C) register.  $B_1$  shifts to  $B_0$ , etc.  $B_7$  shifts into  $B_6$ . However,  $B_7$  itself remains unchanged. Figure 6-10 illustrates the execution of the ASRB instruction. Notice that there are also ASRA and ASR instructions listed in Figure 6-6. These perform the same type of shift operation but on accumulator A and the selected memory byte respectively.

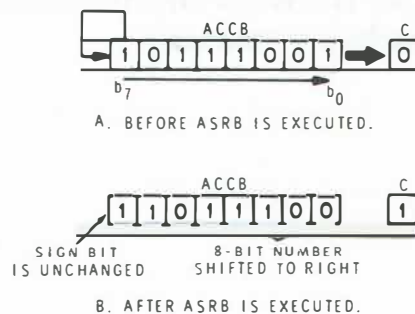


Figure 6-10

Executing the ASRB instruction.



The logic shift-right instructions are different in that they do not retain the sign bit. When a logic shift-right is executed, the contents of the affected register are shifted to the right. The LSB goes into the carry register, and the MSB is filled with a 0. The LSR instruction is illustrated in Figure 6-11. While this instruction shifts the selected memory locations, LSRA and LSRB can be used to perform similar operations on accumulators A and B respectively.

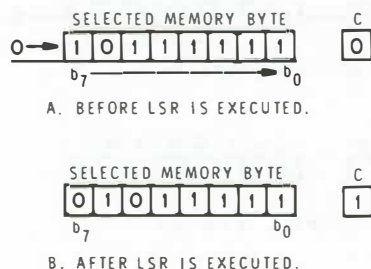


Figure 6-11

Executing the LSR instruction.

Referring back to Figure 6-6, the store accumulator instructions are self-explanatory.

The final data handling instructions are the transfer accumulator instructions. TAB copies the contents of accumulator A into accumulator B. After this instruction is executed, the number originally in accumulator A will now be in both accumulators.

TBA does just the opposite. It copies the contents of accumulator B into accumulator A. After TBA is executed, the number originally in accumulator B will be in both accumulators.

## Logic Instructions

The logic instructions in the 6808 MPU are similar to those in our hypothetical MPU. Figure 6-12 shows the 6808's logic instructions.

There is one AND instruction for each accumulator. The contents of the specified accumulator are ANDed bit-for-bit with the contents of the selected memory location. The result is placed back in the accumulator. This is identical to the AND instruction in our hypothetical machine.

The complement instructions allow you to take the 1's complement of the number in the affected register. COM allows you to complement a byte in memory.

COMA and COMB allow you to complement the contents of accumulators A and B respectively. In each case, all 1's are changed to 0's and all 0's are changed to 1's.

The exclusive OR instruction works like the one in our hypothetical MPU. The contents of the specified accumulator are exclusively ORed bit-for-bit with the contents of the selected memory location. The result is stored back in the specified accumulator.

The inclusive OR is similar except that the contents of the specified accumulator are inclusively ORed with the contents of the selected memory location.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
OPERATIONS	MNEMONIC		H	I	N	Z	V	C
And	ANDA	$A \bullet M \rightarrow A$	●	●	↑	↑	R	●
	ANDB	$B \bullet M \rightarrow B$	●	●	↑	↑	R	●
Complement, 1's	COM	$\bar{M} \rightarrow M$	●	●	↑	↑	R	S
	COMA	$\bar{A} \rightarrow A$	●	●	↑	↑	R	S
	COMB	$\bar{B} \rightarrow B$	●	●	↑	↑	R	S
			●	●	↑	↑	R	S
Exclusive OR	EORA	$A \oplus M \rightarrow A$	●	●	↑	↑	R	●
	EORB	$B \oplus M \rightarrow B$	●	●	↑	↑	R	●
Or, Inclusive	ORA	$A \uparrow M \rightarrow A$	●	●	↑	↑	R	●
	ORB	$B \uparrow M \rightarrow B$	●	●	↑	↑	R	●

Figure 6-12

Logic instructions.

## Data Test Instructions

These are a powerful group of instructions that allow us to compare operands in several different ways. In previous units, you had experience comparing operands; the most frequently-used method was to subtract one operand from another and test the result for zero or negative. In many cases, the numeric result of the subtraction was unimportant. We simply needed to know if the result was zero or minus. The data test instructions allow us to make several different tests without actually producing an unwanted numeric result. These instructions are shown in Figure 6-13.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Bit Test	BITA	A • M	●	●	↑	↑	R	●
	BITB	B • M	●	●	↑	↑	R	●
Compare	CMPA	A – M	●	●	↑	↑	↑	↑
	CMPB	B – M	●	●	↑	↑	↑	↑
Compare Acmltrs	CBA	A – B	●	●	↑	↑	↑	↑
Test, Zero or Minus	TST	M – 00	●	●	↑	↑	R	R
	TSTA	A – 00	●	●	↑	↑	R	R
	TSTB	B – 00	●	●	↑	↑	R	R

Figure 6-13

Data test instructions.

The bit test instructions are very similar to the AND instructions. In both cases, the contents of a specified accumulator are ANDed with the contents of the selected memory location. The difference is that, with the bit test instruction, no logical product is produced. Neither the contents of the accumulator nor memory are altered in any way. However, the condition code registers are affected just as if the AND operation had taken place. Consider the BITA instruction. When executed, A is ANDed with M. If the result is  $00_{16}$ , the (Z) register is set; otherwise, the (Z) register is cleared. If the MSB of the result is 1, the (N) flag is set. However, the contents of the accumulator and memory are unaffected.

In the same way, the compare instructions are similar to subtract instructions except that the resulting numeric difference is ignored. For example, when the CMPA instruction is executed, the contents of the selected memory location are subtracted from the contents of accumulator A. The condition codes are affected just as if a difference had been produced. Again, the contents of accumulator A and memory are unaffected.

The compare accumulators instruction (CBA) works the same way. The condition codes are set if the contents of accumulator B were subtracted from the contents of accumulator A. The contents of the accumulators are unaffected.

Finally, the test for zero or minus instruction allow you to test the number in one of the accumulators or the memory to see if it is negative or zero. When this instruction is executed, the MPU looks at the number in question and sets the (N) and (Z) flags accordingly. The number itself is not changed.

## Index Register and Stack Pointer Instructions

The index register and the stack pointer are 16-bit registers. Figure 6-14 shows eleven instructions that allow us to control the operation of these registers. Because of the 16-bit format, the load, store, and compare instructions are slightly different from those discussed earlier.

INDEX REGISTER AND STACK			5	4	3	2	1	0
POINTER OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION	H	I	N	Z	V	C
Compare Index Reg	CPX	$(X_H/X_L) - (M/M + 1)$	•	•	①	↑	②	•
Decrement Index Reg	DEX	$X - 1 \rightarrow X$	•	•	•	↑	•	•
Decrement Stack Pntr	DES	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX	$X + 1 \rightarrow X$	•	•	•	↑	•	•
Increment Stack Pntr	INS	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	③	↑	R	•
Load Stack Pntr	LDS	$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	③	↑	R	•
Store Index Reg	STX	$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	③	↑	R	•
Store Stack Pntr	STS	$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	③	↑	R	•
Indx Reg $\rightarrow$ Stack Pntr	TXS	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr $\rightarrow$ Indx Reg	TSX	$SP + 1 \rightarrow X$	•	•	•	•	•	•

- ① (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?  
 ② (Bit V) Test: 2's complement overflow from subtraction of LS bytes?  
 ③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

Figure 6-14

Index register and stack pointer instructions.

The compare index register (CPX) instruction allows us to compare the 16-bit number in the index register with any two consecutive bytes in memory. Recall that the index register (X) will hold two bytes. The higher byte is identified as  $X_H$  while the lower byte is called  $X_L$ . When the CPX instruction is executed,  $X_H$  is compared with the 8-bit byte in the specified memory location (M). Also,  $X_L$  is compared with the byte immediately following the specified memory location (M + 1). The comparison is the same as if M and M + 1 were subtracted from  $X_H$  and  $X_L$  except that no numeric difference is produced. Neither X nor M is changed in any way. However, the (N), (Z), and (V) condition codes are affected as shown in Figure 6-14. Generally, the (Z) code is the one we are interested in, since it tells us whether or not an exact match exists between the index register and the two bytes in memory.



The next four instructions are self-explanatory. They allow us to increment and decrement either the index register or the stack pointer. For one thing, these instructions allow us to maintain two separate 16-bit tallies simultaneously. However, the real value of these instructions and their associated registers will be discussed later.

The load and store instructions for the 16-bit registers are shown next in Figure 6-14. Since these are 2-byte registers, the LDX and LDS instructions must load two bytes from memory. In the case of the index register, the specified memory byte ( $M$ ) is loaded into the upper half of the index register ( $X_H$ ). An instant later, the next byte in memory ( $M + 1$ ) is automatically loaded into the lower half of the index register ( $X_L$ ). Thus, the operation can be described as:  $M \rightarrow X_H, (M + 1) \rightarrow X_L$ .

Because the stack pointer is also a 16-bit register, the load stack pointer instruction (LDS) works the same way. Its operation can be described as:  $M \rightarrow SP_H, (M + 1) \rightarrow SP_L$ . Here,  $SP_H$  refers to the upper half of the stack pointer, while  $SP_L$  refers to the lower half.

When the contents of the 16-bit registers are being stored, the operation is reversed. For example, the STX instruction stores  $X_H$  in  $M$  and  $X_L$  in  $M + 1$ . A similar instruction, STS, allows us to store the contents of the stack pointer in the same way.

The final two instructions in this group allow us to transfer numbers between these two 16-bit registers. The TXS instruction loads the stack pointer with the contents of the index register **minus one**. The TSX instruction loads the index register with the contents of the stack pointer **plus one**. A more detailed discussion of these two important registers and their associated instructions will be given towards the end of this unit.

## Branch Instructions

The branch instructions are shown in Figure 6-15. Two additional instructions are also included since they affect the program counter.

BRANCH			5	4	3	2	1	0
OPERATIONS	MNEMONIC	BRANCH TEST	H	I	N	Z	V	C
Branch Always	BRA	None	•	•	•	•	•	•
Branch If Carry Clear	BCC	$C = 0$	•	•	•	•	•	•
Branch If Carry Set	BCS	$C = 1$	•	•	•	•	•	•
Branch If = Zero	BEQ	$Z = 1$	•	•	•	•	•	•
Branch If $\geq$ Zero	BGE	$N \oplus V = 0$	•	•	•	•	•	•
Branch If $>$ Zero	BGT	$Z + (N \oplus V) = 0$	•	•	•	•	•	•
Branch If Higher	BHI	$C + Z = 0$	•	•	•	•	•	•
Branch If $\leq$ Zero	BLE	$Z + (N \oplus V) = 1$	•	•	•	•	•	•
Branch If Lower Or Same	BLS	$C + Z = 1$	•	•	•	•	•	•
Branch If $<$ Zero	BLT	$N \oplus V = 1$	•	•	•	•	•	•
Branch If Minus	BMI	$N = 1$	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	$Z = 0$	•	•	•	•	•	•
Branch If Overflow Clear	BVC	$V = 0$	•	•	•	•	•	•
Branch If Overflow Set	BVS	$V = 1$	•	•	•	•	•	•
Branch If Plus	BPL	$N = 0$	•	•	•	•	•	•
No Operation	NOP	Advances Prog. Cntr. Only	•	•	•	•	•	•
Wait for Interrupt	WAI		•	①	•	•	•	•

- ① (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

Figure 6-15

Jump and branch instructions.

Nine of these instructions were discussed in the previous unit. These are:

- Branch Always (BRA).
- Branch If Carry Clear (BCC).
- Branch If Carry Set (BCS).
- Branch If Equal Zero (BEQ).
- Branch If Not Equal Zero (BNE).
- Branch If Minus (BMI).
- Branch If Plus (BPL).
- Branch If Overflow Clear (BVC).
- Branch If Overflow Set (BVS).

Before we discuss the new branch instructions, here are some of the symbols we will be using. The symbol ( $\geq$ ) means “is greater than or is equal to; ( $>$ ) means “is greater than; ( $\leq$ ) means “is less than or equal to; ( $<$ ) means “is less than”; and ( $\neq$ ) means “is not equal to.”

Now consider the Branch-If-Greater-Than-or-Equal instruction (BGE). This instruction is normally used after a subtract or compare instruction. It will cause a branch operation if the two's complement value in the accumulator is greater than or equal to the two's operand in memory. This condition is indicated by the (N) and (V) flags having the same value. The MPU determines if this condition is met by exclusively ORing (N) and (V) and examining the result.

Three simple examples may help illustrate the operation of this instruction. Let's start with a number in the accumulator that is greater than the operand in memory.

Number in Accumulator	=	00000010 <sub>2</sub>
Operand in Memory	=	00000001 <sub>2</sub>

When the operand is subtracted, the result is 00000001<sub>2</sub>. With this result, both (N) and (V) are cleared to 0. Notice that (N) and (V) are equal and  $N \oplus V = 0$ . If the BGE instruction followed the subtract operation, the branch would be implemented.

Now see what happens when the number in the accumulator is equal to the operand in memory:

Number in Accumulator = 00000010<sub>2</sub>  
Operand in Memory = 00000010<sub>2</sub>

When the operand is subtracted, the result is 00000000<sub>2</sub>. Again, (N) and (V) are cleared to 0. Thus (N) and (V) are still equal and  $N \oplus V = 0$ . Again, the BGE instruction would cause a branch to occur.

Finally, note what happens when the number in the accumulator is smaller than the operand in memory:

Number in Accumulator = 00000001<sub>2</sub>  
Operand in Memory = 00000010<sub>2</sub>

When the operand is subtracted, the result is 11111111<sub>2</sub>. This time (N) is set, but (V) is cleared. Thus (N) and (V) are not equal. Therefore,  $N \oplus V = 1$ . In this case, the BGE conditions are not equal, and no branch will occur. The branch occurs if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory.

Next, consider the Branch-If-Greater-Than (BGT) instruction. This instruction is normally used immediately after a subtract or compare operation. The branch will occur only if the two's complement minuend was greater than the two's complement subtrahend. By trying several examples as we did above, you will find that the branch conditions are met when  $Z = 0$  and  $N = V$ .

The Branch-If-Higher (BHI) instruction is similar to the BGT instruction, except that it is concerned with **unsigned** numbers. BHI is normally used after a subtract or compare operation. The branch will occur only if the unsigned minuend was greater than the unsigned subtrahend. By trying several different examples, you can prove that this occurs only when the (C) and (Z) flags are both 0.

The Branch-If-Less-Than-or-Equal (BLE) instruction allows you to compare two's complement numbers in another way. If it is executed immediately after a subtract or compare operation, the branch will occur only if the two's complement minuend was less than or equal to the two's complement subtrahend.

The Branch-If-Lower-Or-Same (BLS) instruction is similar to the BLE instruction, except that **unsigned** numbers are compared. When it is executed immediately after a subtract or compare operation, the branch will occur if the unsigned minuend was lower than or equal to the unsigned subtrahend.

The Branch-If-Less-Than-Zero (BLT) instruction is also similar to the BLE instruction, except that the equal qualification is removed. If BLT is executed immediately after a subtract or compare operation, the branch occurs only if the two's complement minuend was less than the two's complement subtrahend.

Two additional instructions are included in Figure 6-15. Although they are not branch instructions, they are included here since they do not seem to fit any of the other categories.

The No-Operation (NOP) instruction is a "do-nothing" instruction that simply consumes a small increment of time. It does not change the contents of any register except the program counter. It does increment the program counter by one and consumes two MPU cycles. In spite of this, the NOP is a very useful instruction. When writing a program, we frequently use too many instructions. Once the program is loaded in memory, it is inconvenient to remove an unwanted instruction. If you remove an instruction, a hole is left in memory, and it becomes necessary to move back all following instructions to fill the hole. It is much easier to fill the hole with one or more NOP instructions.

The Wait-For-Interrupt (WAI) instruction is the 6808's version of a HLT instruction. Previously, we used this instruction at the end of all our programs. We will continue to use it in the same manner in the future. However, as you will see later in this unit, there is more involved in executing the WAI instruction than simply stopping the MPU. For now, though, continue to think of the WAI as a simple halt instruction.



## Condition-Code-Register Instructions

The 6808 MPU has eight instructions that allow us direct access to the condition codes. They are listed in Figure 6-16.

CONDITION CODE REGISTER			5	4	3	2	1	0
OPERATIONS	MNEMONIC	BOOLEAN OPERATION	H	I	N	Z	V	C
Clear Carry	CLC	$0 \rightarrow C$	•	•	•	•	•	R
Clear Interrupt Mask	CLI	$0 \rightarrow I$	•	R	•	•	•	•
Clear Overflow	CLV	$0 \rightarrow V$	•	•	•	•	R	•
Set Carry	SEC	$1 \rightarrow C$	•	•	•	•	•	S
Set Interrupt Mask	SEI	$1 \rightarrow I$	•	S	•	•	•	•
Set Overflow	SEV	$1 \rightarrow V$	•	•	•	•	S	•
Accmltr A $\rightarrow$ CCR	TAP	$A \rightarrow CCR$	①					
CCR $\rightarrow$ Accmltr A	TPA	$CCR \rightarrow A$	•	•	•	•	•	•

R = Reset

S = Set

• = Not affected

① (ALL) Set according to the contents of Accumulator A.

Figure 6-16

Condition code register instructions.

Most of these instructions are self-explanatory. The Clear-Carry (CLC) instruction resets the (C) flag to 0 while the Set-Carry (SEC) instruction sets it to 1. In the same manner, the CLV and SEV instructions allow us to clear and set the overflow (V) flag. Also, the CLI and SEI instructions can be used to clear or set the interrupt (I) flag.

You will notice that there are no instructions for individually clearing the (N), (Z), or (H) flags. However, we can still set or clear these flags with the Transfer-Accumulator-A-To-The-Processor-Condition-Code-Register (TAP) instruction. Figure 6-17 illustrates the execution of this instruction. The contents of bits 0 through 5 of accumulator A are transferred to the condition code registers. Thus, this instruction allows us to set or clear all the condition codes at once.

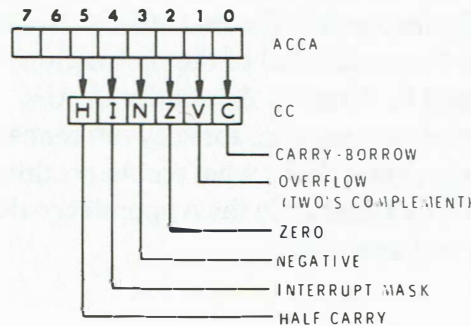


Figure 6-17

Executing the TAP instruction.

The final instruction is the Transfer-From-The-Processor-Condition-Codes-Register-To-Accumulator-A (TPA) instruction. When this instruction is executed, the contents of the condition code registers are transferred to bits 0 through 5 of accumulator A. This operation is illustrated in Figure 6-18. Note that bits 6 and 7 of the accumulator are set to 1.

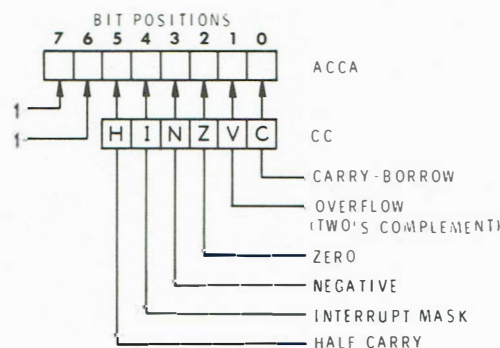


Figure 6-18

Executing the TPA instruction.

## Summary of Instruction Set

As you can see, the 6808 MPU has a wide variety of instructions. In this section, most of the instructions have been mentioned briefly. However, a full explanation of some instructions must wait until new concepts have been covered.

In one short section, it is very difficult to cover every instruction in detail. Also, it is virtually impossible for the reader to remember all the details of each instruction. Remember, all of the instructions available to the 6808 MPU are explained in detail in **Appendix C**. Also, they are arranged alphabetically by their mnemonics for easy reference. Refer to **Appendix C** any time you are in doubt about what an instruction does. Be sure to look over the introductory material in the Appendix so that you understand all the conventions and symbols.

## Programmed Review

12. You will find a detailed explanation of all 6808 MPU instructions in \_\_\_\_\_ of this course.
13. (Appendix C) In an operation represented by the shorthand notation  $A + B \rightarrow A$ , the sum of the operation would be transferred into \_\_\_\_\_.
14. (accumulator A) If the result of an arithmetic operation is zero, the (Z) flag \_\_\_\_\_ set to 1.  
(is/is not)
15. (is) Regarding the 6808 MPU, the \_\_\_\_\_ instruction allows you to maintain a tally in memory without having to load it, increment it, and then store it.
16. (increment/INC) The decimal adjust instruction works only with the \_\_\_\_\_ accumulator.  
(A/B)
17. (A) When the RORA instruction is executed, the LSB of accumulator A is shifted into the \_\_\_\_\_ register.
18. (carry (C)) Once the TAB instruction has been executed and the contents of accumulator A has been transferred to accumulator B, the contents originally in accumulator A \_\_\_\_\_ be in both accumulators.  
(will/will not)
19. (will) Data test instructions are used to \_\_\_\_\_ various operands.
20. (compare) The index register and stack pointer are \_\_\_\_\_-bit registers.  
(8/16)

21. (16) The compare-index-register (CPX) instruction allows you to compare the 16-bit number in the index register with two \_\_\_\_\_ bytes in memory.

(random/consecutive)

22. (consecutive) The Branch-If-Higher (BHI) instruction is concerned only with \_\_\_\_\_ numbers.

(signed/unsigned)

23. (unsigned) The \_\_\_\_\_ instruction is a “do-nothing” instruction that simply consumes a small increment of time.

24. (No-Operation/NOP) The NOP instruction consumes \_\_\_\_\_ MPU cycle(s).

(one/two)

25. (two) When the \_\_\_\_\_ instruction is executed, the contents of the condition code registers are transferred to bits 0 through 5 of accumulator A.

(TPA/TAP)

(TPA)



## NEW ADDRESSING MODES

Previously, we discussed four addressing modes. Let's briefly review these.

In the immediate addressing mode, the operand is the memory byte immediately following the opcode. These are generally 2-byte instructions. The first byte is the opcode; the second is the operand. However, there are some exceptions to the 2-byte rule. Some operations involve the 16-bit register and stack pointer. In these cases, the operand is the two bytes immediately following the opcode. These are 3-byte instructions. The first byte is the opcode, and the second and third bytes are the operand.

In the direct addressing mode, the byte following the opcode is the address of the operand. These are always 2-byte instructions. The first byte is the opcode; the second is the address of the operand. An 8-bit byte can specify addresses from  $00$  to  $FF_{16}$ . Thus, when the direct addressing mode is being used, the operand must be in the first  $256_{10}$  bytes of memory. Since the 6808 MPU can have up to  $65,536_{10}$  bytes of memory, another means must be used to address the upper portion of memory.

The relative addressing mode is used for branching. These are 2-byte instructions. The first byte is the opcode; the second is the relative address. Recall that the relative address is added to the program count to form the absolute address. Since the 8-bit relative address is a two's complement number, the branch limits are  $+127_{10}$  and  $-128_{10}$ .

In the inherent addressing mode, there is either no operand or the operand is implied by the instruction. These are 1-byte instructions.

In this section, we will discuss two new addressing modes. These are called **extended addressing** and **indexed addressing**. We will discuss extended addressing first.

## Extended Addressing

Extended addressing is similar to direct addressing but with one significant difference. Recall that with direct addressing, the operand must be in the first  $256_{10}$  bytes of memory. Since this represents less than one percent of the addresses available to the 6808 MPU, a more powerful addressing mode is required. The extended addressing mode fills this requirement.

The format of an instruction that uses extended addressing is shown in Figure 6-19. The instruction will always have three bytes. The first byte is the opcode. The second and third bytes form a 16-bit address. Notice that the most significant part of the address is the byte immediately following the opcode. Since this instruction has a 16-bit address, the operand can be any one of the  $65,536_{10}$  possible addresses.

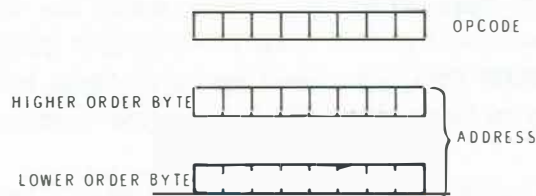


Figure 6-19

Format of an instruction that uses the extended addressing mode.

Suppose for example, that you wish to load the operand at memory location  $2134_{16}$  into accumulator B. The instruction would look like this:

F6 Opcode for LDAB extended.  
 21 Higher order address.  
 34 Lower order address.

By the same token, if you wish to increment the number in memory location  $AA00_{16}$ , the instruction would be:

7C Opcode for INC extended.  
 AA Higher order address.  
 00 Lower order address.

The extended addressing mode allows us to address an operand at any address including the first  $256_{10}$  bytes of memory. Thus, if you wish to load the operand at address  $0013_{16}$  into accumulator A, you can use extended addressing:

B6	Opcode for LDAA extended.
00	Higher order address.
13	Lower order address.

Or, you can use direct addressing:

96	Opcode for LDAA direct.
13	Address.

Notice that, with direct addressing, the higher order address can be ignored since it is always 00. Because it saves one memory byte and one MPU cycle, direct addressing is normally used when the operand is in the first  $256_{10}$  bytes of memory. Extended addressing is used when the operand is above address  $00FF_{16}$ . However, as you will see later, some instructions do not have a direct addressing mode. In these cases, extended addressing must be used even if the operand is in the first  $256_{10}$  memory locations.

## Indexed Addressing

The most powerful addressing mode available to the 6808 MPU is indexed addressing. Recall that the 6808 MPU has a 16-bit index register. There are several instructions associated with this register that allow us to load the register from memory and to store its contents in memory. Also, we can increment and decrement the index register. We can even compare its contents with two consecutive bytes in memory. These capabilities alone make the index register a very handy 16-bit counter. However, the real power of the index register comes from the fact that we can use this counter as an index pointer. Since this is a 16-bit register, it can point to any address in memory.

**Purpose**—Before going into the detail of how indexed addressing works, let's see why it is needed. Let's assume that we wish to add a list of  $20_{16}$  numbers, and that the numbers are in  $20_{16}$  consecutive address locations starting at address 0050. Using the addressing modes discussed earlier, our program might look like this:

CLRA	Clear Accumulator A.
ADDA	Add the first number
50	to Accumulator A.
ADDA	Add the second number
51	to Accumulator A.
ADDA	Add the third number
52	to Accumulator A.
*	*
*	*
*	*
*	*
ADDA	Add the last number
6F	to Accumulator A.
WAI	Wait.

While this accomplishes the desired result, it requires a long, repetitive program. The above program would require  $66_{10}$  bytes of memory. Notice that all the ADDA instructions are identical except that each successive address is one larger than the previous address. Indexed addressing can greatly simplify programs of this type.

**Instruction Format**—The format of an instruction that uses indexed addressing is shown in Figure 6-20. Notice that this is a 2-byte instruction. The first byte is the opcode, and the second is called an offset address. The offset address is an **unsigned** 8-bit binary number. It is added to the contents of the index register to determine the address at which the operand is located.

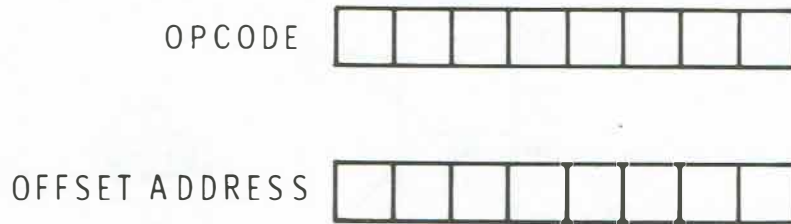


Figure 6-20

Format of an instruction that used  
the indexed addressing mode.

Every instruction that involves an operand in memory can use the indexed addressing mode. In this unit, we will use the following convention to indicate indexed addressing:

LDAA, X  
STAA, X  
ADDB, X  
etc.

In each case, the X tells us that indexed addressing is used. For example, the first instruction means: “using indexed addressing, load the contents of the specified memory location into accumulator A.” Now let’s see how the address of the operand is determined.



**Determining the Operand Address**—When indexed addressing is being used, the address of the operand is determined by the offset address and the number in the index register. Specifically, the 8-bit offset address is added to the 16-bit address in the index register. The 16-bit sum becomes the address of the operand. Figure 6-21 illustrates this.

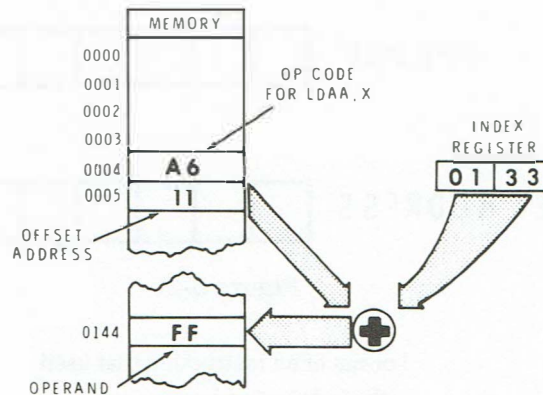


Figure 6-21

The operand address is formed by adding the offset address to the contents of the index register.

Here, the instruction in memory location  $0004_{16}$  is LDAA, X. The offset address is  $11_{16}$ . The contents of the index register are  $0133_{16}$ . When the LDAA, X instruction is executed, the address of the operand is formed by adding the offset address to the number in the index register. In this case, the operand address will be:

$$\begin{array}{r} 0133_{16} \\ + 11_{16} \\ \hline 0144_{16} \end{array}$$

The operand of this address is loaded into accumulator A. In this example, the operand FF is loaded into accumulator A when the instruction at location 0004 is executed. It is important to remember that this does not change the contents of the index register in any way. That is, the index register will still contain  $0133_{16}$  after the instruction is executed.

**Adding A List of Numbers**—To see how this addressing mode saves instructions, consider the problem given earlier. Recall that we were to add  $20_{16}$  numbers stored in consecutive memory locations starting at address 0050. Using indexed addressing for the add instruction, our program looks like the one shown in Figure 6-22.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with the
0011	00	00	address of the
0012	50	50	first number in list.
0013	4F	CLRA	Clear accumulator A
0014	AB	→ ADDA X	Add to accumulator A using indexed addressing
0015	00	00	with an offset address of 00.
0016	08	INX	Increment index register.
0017	8C	CPX #	Compare the contents of the index register
0018	00	00	with an address that is one greater than the
0019	70	70	address of the last number in the list.
001A	26	BNE	If not equal, branch back
001B	F8	F8	to the ADDA, X instruction.
001C	3E	WAI	Otherwise, halt.

Figure 6-22

Program for adding a list of  $20_{16}$  numbers.

The first instruction is load index register immediate. Notice that a new symbol is used in this program. The symbol # is used to indicate the immediate addressing mode. Thus, the LDA# instruction causes the operand immediately following the opcode to be loaded into the index register. Remember, the index register can hold two 8-bit bytes. The operand is the 2-byte number  $0050_{16}$ . You may recognize that this is the address of the first number in the list of numbers to be added.

The next instruction (CLRA) clears accumulator A. The sum will be accumulated in this register, so it is important that it be cleared initially.

The third instruction (ADDA, X) is the only instruction in the program that uses indexed addressing. Notice that the symbol X indicated the indexed addressing mode. The offset address is 00. Recall that the operand address is determined by adding the offset to the contents of the index register. The index register contains  $0050_{16}$  from a previous instruction. Since the offset is 00, the operand is  $0050_{16}$ . That is, the contents of memory location 0050 are added to the contents of accumulator A. Recall that  $0050_{16}$  is the address of the first number in the list.

The fourth instruction increments the index register to  $0051_{16}$ . Notice that the index register now points to the address of the second number in the list.

The fifth instruction compares the number in the index register with a number that is one greater than the address of the last number in the list.

If a match occurs, the (Z) flag will be set. Of course in this case, no match occurs yet. Notice once again that the symbol # indicates the immediate addressing mode. Thus, the contents of the index register are compared with the next two bytes in the program or  $0070$ .

The BNE instruction tests the (Z) flag to see if the two numbers matched. If no match is indicated, the relative address (F8) directs the program back to the ADDA, X instruction. The first path through the loop ends with the first number in accumulator A.

The second pass through the loop begins with the ADDA, X instruction being executed again. This time however, the index register points to address  $0051$ . Therefore, the second number in the list is added to accumulator A. Accumulator A now contains the sum of the first two numbers. The index register is then incremented to  $0052$ , and its contents are again compared with  $0070$ . No match exists, so the BNE instruction causes the loop to be repeated once again.

The loop is repeated over and over. Each time, the next number in the list is added to the contents of accumulator A. This process continues until the last number in the list is added. At that time, the index register will be incremented to  $0070$ . Thus, when the CPX# instruction is executed, the (Z) flag will be set because the two numbers match. The BNE instruction recognizes that a match has occurred. Consequently, it does not allow the branch to occur and the next instruction in sequence is executed. Because this is the WAI instruction, the program halts. At this time, the sum of the  $20_{16}$  numbers in the list will be in accumulator A.

Adding a list of numbers is a classic example of how indexing can be used to shorten a program. However, this example does not illustrate the full power of the indexed addressing. For example, it does not illustrate the advantage of the offset address. Because indexed addressing is so important, let's look at another example.

**Copying A List**—Let's assume we have a list of  $10_{16}$  numbers that we wish to copy from one location to another. For simplicity, assume that the list is presently in addresses 0030 through 003F and that we wish to copy the list in location 0040 through 004F. Without using indexed addressing, our program might look like this:

```
LDAA
 30
STAA
 40
LDAA
 31
STAA
 41
*
*
*
LDAA
 3F
STAA
 4F
WAI
```

As you have seen; long, repetitive programs such as this are excellent candidates for indexed addressing.

Using indexed addressing, our program might look like that shown in Figure 6-23. The first step is to load the index register with the first address in the original list. The LDAA, X instruction has an offset address of 00. Therefore, accumulator A is loaded from the address specified by the index register (0030). That is, the first number in the original list is loaded into accumulator A when the LDAA, X instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with
0011	00	00	the first address of the original
0012	30	30	list.
0013	A6	LDAA, X	Load accumulator A indexed with
0014	00	00	an offset of 00.
0015	A7	STAA, X	Store accumulator A indexed with
0016	10	10	an offset of 10, i.e.,
0017	08	INX	Increment index register.
0018	8C	CPX #	Compare index with one greater
0019	00	00	than last
001A	40	40	address in original list.
001B	26	BNE	If not equal, branch back to the
001C	F6	F6	LDAA, X instruction.
001D	3E	WAI	Otherwise, halt.

Figure 6-23

Program for copying a list from addresses 0030 — 003F into  
addresses 0040 — 004F.

The STAA,X instruction illustrates the use of the offset address. Notice that the offset is 10. This number is added to the address in the index register to form the effective address at which the contents of accumulator A are stored. Thus, the contents of accumulator A are stored at address 0040. Remember, this does not change the number in the index register in any way. By using the offset, we can load the accumulator indexed from one address and store the accumulator indexed at another.

Next, the index register is incremented to 0031. It is then compared with 0040. Since no match exists, the BNE instruction directs the program back to the LDAA, X instruction. The loop is repeated until the entire list is rewritten in locations 0040 through 004F. After the last entry in the list is copied, the index register is incremented to 0040. Thus, the CPX# instruction sets the (Z) flag, allowing the BNE instruction to divert the program from the loop. The program halts after the last entry in the list is written in its new position in memory.



## Instruction Set Summary

You have been introduced to most of the instructions available to the 6808 MPU. You have also been introduced to all the addressing modes. Now let's look at the complete instruction set.

Figure 6-24 summarizes the 6808's instructions and addressing modes. This 2-page Figure contains a wealth of information. For your convenience, this information is repeated on the **Instruction Set Summary card** provided with the course. You should keep this card handy. After a while, you will be able to write long, complex programs using only the card for reference.

The left-hand column of Figure 6-24 lists the names and mnemonics for each of the instructions. In many cases, a single name such as "add" is associated with more than one mnemonic. For example, ADDA is an add operation that involves accumulator A while ADDB is an add operation that involves accumulator B.

The center column gives important information about the addressing modes. Notice that the ADDA instruction can have any one of four addressing modes: immediate, direct, indexed, or extended. Three facts are given for each addressing mode. The hexadecimal opcode is given in the OP column. For example, the opcode for ADDA direct is 9B.

The column labeled (~) tells the number of MPU cycles required to execute the instruction. This information is important because it allows us to determine exactly how long it will take to run a given program. An MPU cycle is equal to one cycle of the MPU clock. For example, if the clock frequency is 1 MHz, one MPU cycle will be one microsecond. With this clock rate, 2 microseconds are required to execute the ADDA immediate instruction while 5 microseconds are required for the ADDA indexed instruction.

The column labeled (#) indicates the number of bytes required by the instruction. ADDA immediate, ADDA direct, and ADDA indexed are 2-byte instructions, while ADDA extended is 3-byte instruction.

The next column to the right gives the shorthand notation for the Boolean or arithmetic operations performed. Finally, the right-hand indicates how the condition code registers are affected by each instruction.

ACCUMULATOR AND MEMORY		ADDRESSING MODES															COND. CODE REG.					
		IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)					
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#						
OPERATIONS	MNEMONIC																					
Add	ADDA	88	2	2	98	3	2	A8	5	2	B8	4	3				A + M → A					
	ADD8	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B + M → B					
Add Acmltrs	ABA													1B	2	1	A + B → A					
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A					
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B					
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A					
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B					
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M					
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M					
Clear	CLR							6F	7	2	7F	6	3				00 → M					
	CLRA													4F	2	1	00 → A					
	CLRB													5F	2	1	00 → B					
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A - M					
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B - M					
Compare Acmltrs	CBA													11	2	1	A - B					
Complement, 1's	COM							63	7	2	73	6	3				M → M					
	COMA													43	2	1	A → A					
	COMB													53	2	1	B → B					
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 - M → M				①	②
	NEGA													40	2	1	00 - A → A				①	②
	NEGB													50	2	1	00 - B → B				①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format					③
Decrement	DEC							6A	7	2	7A	6	3				M - 1 → M				④	
	DECA													4A	2	1	A - 1 → A				④	
	DECB													5A	2	1	B - 1 → B				④	
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A					
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B					
Increment	INC							6C	7	2	7C	6	3				M + 1 → M				⑤	
	INCA													4C	2	1	A + 1 → A				⑤	
	INCB													5C	2	1	B + 1 → B				⑤	
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A					
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B					
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A ∨ M → A					
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B ∨ M → B					
Push Data	PSHA													36	4	1	A → M <sub>SP</sub> , SP - 1 → SP					
	PSHB													37	4	1	B → M <sub>SP</sub> , SP - 1 → SP					
Pull Data	PULA													32	4	1	SP + 1 → SP, M <sub>SP</sub> → A					
	PULB													33	4	1	SP + 1 → SP, M <sub>SP</sub> → B					
Rotate Left	ROL							69	7	2	79	6	3				M				⑥	
	ROLA													49	2	1	A				⑥	
	ROLB													59	2	1	B				⑥	
Rotate Right	ROR							66	7	2	76	6	3				M				⑥	
	RORA													46	2	1	A				⑥	
	RORB													56	2	1	B				⑥	
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M				⑥	
	ASLA													48	2	1	A				⑥	
	ASLB													58	2	1	B				⑥	
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M				⑥	
	ASRA													47	2	1	A				⑥	
	ASRB													57	2	1	B				⑥	
Shift Right, Logic	LSR							64	7	2	74	6	3				M				⑥	
	LSRA													44	2	1	A				⑥	
	LSRB													54	2	1	B				⑥	
Store Acmltr	STAA				97	4	2	A7	6	2	B7	5	3				A → M					
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M					
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A - M → A					
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B - M → B					
Subtract Acmltrs	SBA													10	2	1	A - B → A					
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A - M - C → A					
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B - M - C → B					
Transfer Acmltrs	TAB													16	2	1	A → B					
	TBA													17	2	1	B → A					
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M - 00					
	TSTA													4D	2	1	A - 00					
	TSTB													5D	2	1	B - 00					

Figure 6-24  
The 6808 instruction set

INDEX REGISTER AND STACK		IMMED			DIRECT			INDEX			EXTND			INNER			BOOLEAN/ARITHMETIC OPERATION	5 4 3 2 1 0											
POINTER OPERATIONS		MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	H	I	N	Z	V	C						
Compare Index Reg	CPX		8C	3	3	9C	4	2	AC	6	2	BC	5	3															
Decrement Index Reg	DEX														09	4	1												
Decrement Stack Pntr	D ES														34	4	1												
Increment Index Reg	INX														08	4	1												
Increment Stack Pntr	INS														31	4	1												
Load Index Reg	LDX		CE	3	3	DE	4	2	EE	6	2	FE	5	3															
Load Stack Pntr	LDS		8E	3	3	9E	4	2	AE	6	2	BE	5	3															
Store Index Reg	STX					DF	5	2	EF	7	2	FF	6	3															
Store Stack Pntr	STS					9F	5	2	AF	7	2	BF	6	3															
Indx Reg → Stack Pntr	TXS														35	4	1												
Stack Pntr → Indx Reg	TSX														30	4	1												
																		(X <sub>H</sub> /X <sub>L</sub> ) - (M/M + 1)											
																		X - 1 → X											
																		SP - 1 → SP											
																		X + 1 → X											
																		SP + 1 → SP											
																		M → X <sub>H</sub> , (M + 1) → X <sub>L</sub>											
																		M → SP <sub>H</sub> , (M + 1) → SP <sub>L</sub>											
																		X <sub>H</sub> → M, X <sub>L</sub> → (M + 1)											
																		SP <sub>H</sub> → M, SP <sub>L</sub> → (M + 1)											
																		X - 1 → SP											
																		SP + 1 → X											

JUMP AND BRANCH		RELATIVE			INDEX			EXTND			INNER			BRANCH TEST	5 4 3 2 1 0													
OPERATIONS		MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	H	I	N	Z	V	C								
Branch Always	BRA		20	4	2																							
Branch If Carry Clear	BCC		24	4	2																							
Branch If Carry Set	BCS		25	4	2																							
Branch If = Zero	BEQ		27	4	2																							
Branch If ≥ Zero	BGE		2C	4	2																							
Branch If > Zero	BGT		2E	4	2																							
Branch If Higher	BHI		22	4	2																							
Branch If ≤ Zero	BLE		2F	4	2																							
Branch If Lower Or Same	BLS		23	4	2																							
Branch If < Zero	BLT		2D	4	2																							
Branch If Minus	BMI		28	4	2																							
Branch If Not Equal Zero	BNE		26	4	2																							
Branch If Overflow Clear	BVC		28	4	2																							
Branch If Overflow Set	BVS		29	4	2																							
Branch If Plus	BPL		2A	4	2																							
Branch To Subroutine	BSR		8D	8	2																							
Jump	JMP					6E	4	2	7E	3	3																	
Jump To Subroutine	JSR					A0	8	2	BD	9	3																	
No Operation	NOP														01	2	1											
Return From Interrupt	RTI														38	10	1											
Return From Subroutine	RTS														39	5	1											
Software Interrupt	SWI														3F	1	2	1										
Wait for Interrupt	WAI														3E	9	1											
																		See Special Operations										
																		Advances Prog. Cntr. Only										
																		See special Operations										

CONDITIONS CODE REGISTER		INNER			BOOLEAN OPERATION	5 4 3 2 1 0						
OPERATIONS		MNEMONIC	OP	~	#	H	I	N	Z	V	C	
Clear Carry	CLC		0C	2	1	0 → C					R	
Clear Interrupt Mask	CLI		0E	2	1	0 → I		R				
Clear Overflow	CLV		0A	2	1	0 → V				R		
Set Carry	SEC		0D	2	1	1 → C					S	
Set Interrupt Mask	SEI		0F	2	1	1 → I		S				
Set Overflow	SEV		08	2	1	1 → V				S		
Accmltr A → CCR	TAP		06	2	1	A → CCR	12					
CCR → Accmltr A	TPA		07	2	1	CCR → A						

LEGEND:

OP Operation Code (Hexadecimal);  
~ Number of MPU Cycles;  
# Number of Program Bytes;  
+ Arithmetic Plus;  
- Arithmetic Minus;  
• Boolean AND;  
M<sub>SP</sub> Contents of memory location pointed to be Stack Pointer;  
+ Boolean Inclusive OR;  
• Boolean Exclusive OR;  
M Complement of M;  
→ Transfer Into;  
0 Bit = Zero;

00 Byte = Zero;  
H Half-carry from bit 3;  
I Interrupt mask;  
N Negative (sign bit);  
Z Zero (byte);  
V Overflow, 2's complement;  
C Carry from bit 7;  
R Reset Always;  
S Set Always;  
‡ Test and set if true, cleared otherwise;  
• Not Affected;  
CCR Condition Code Register;  
LS Least Significant;  
MS Most Significant

CONDITION CODE REGISTER NOTES:  
(Bit set if test is true and cleared otherwise)  
① (Bit V) Test: Result = 1 0000000?  
② (Bit C) Test: Result = 00000000?  
③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)  
④ (Bit V) Test: Operand = 1 0000000 prior to execution?  
⑤ (Bit V) Test: Operand = 0 11 11 prior to execution?  
⑥ (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred.  
⑦ (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?  
⑧ (Bit V) Test: 2's complement overflow from subtraction of LS bytes?  
⑨ (Bit N) Test: Resu less than zero? (Bit 15 = 1)  
⑩ (All) Load Condition Code Register from Stack. (See Special Operations)  
⑪ (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.  
⑫ (ALL) Set according to the contents of Accumulator A.

Figure 6-24

Figure 6-24

If you study the instruction set carefully, you will find that there are a few instructions that we have not yet discussed. These additional instructions will be discussed after you complete the following Programmed Review.

## EXPERIMENTS

Perform Programming Experiments 9 and 10. You will find these experiments in Unit 12. After you finish these experiments, return to this unit and complete your studies.

## Programmed Review

26.	A disadvantage of direct addressing is that the operand must be in the first _____ bytes of memory.
27.	(256 <sub>10</sub> ) The advantage of direct addressing is that only _____ bytes are required for each instruction.
28.	(two) Extended addressing can address _____ bytes of memory.
29.	(65,536 <sub>10</sub> ) A disadvantage of extended addressing is that each instruction requires _____ bytes.
30.	(three) Extended addressing _____ be used to address an operand in the first 256 <sub>10</sub> bytes of memory. (can/cannot)
31.	(can) The most powerful addressing mode available to the 6808 MPU is called _____ addressing.
32.	(indexed) Indexed addressing requires _____ bytes for each instruction.
33.	(two) The second byte of an indexed addressing instruction is called the _____ address.
34.	(offset) When indexed addressing is used, the address of the operand is determined by adding the offset address to the contents of the _____ register.
	(index)



## STACK OPERATIONS

In computer jargon, a stack is a group of temporary storage locations in which data can be stored and later retrieved. In this regard, a stack is somewhat like memory. In fact, many microprocessors use a section of memory as a stack. The difference between a stack and other forms of memory is the method by which the data is accessed or addressed. The discussion will begin by considering a simple stack arrangement used in some microprocessors. Then, the more sophisticated stack arrangement used by the 6808 MPU will be discussed.

### Cascade Stack

Some microprocessors have a special group of registers, usually 8 or 16, called a **cascade stack**. Each register can hold one 8-bit byte of data. Because these registers are right on the MPU chip, they make excellent temporary storage locations. If we need to free the accumulator for some reason, we can store its contents in the stack. Later, if that piece of data is needed again, we can retrieve the data from the stack. Of course, we could have also freed the accumulator by storing the data in memory. What then is the advantage of the stack?

One advantage of the stack is the method by which it is accessed or addressed. Recall that when a byte is stored in memory, an address is required. That is, to store the contents of the accumulator in memory, a 2-byte or 3-byte instruction is required. Depending on the addressing mode, the last one or two bytes is the address. Later, if the byte is retrieved, another instruction is required that also has an address.

Figure 6-25 shows an 8-register stack similar to that found in some microprocessors. This is called a cascade stack because of the method by which data is loaded and retrieved. All data transfers are between the top of the stack and the accumulator. That is, the accumulator communicates only with the top location on the stack. Data is transferred to the stack by a special instruction called PUSH.

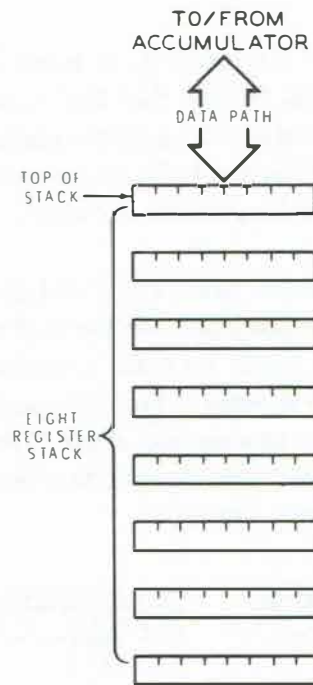


Figure 6-25

A cascade stack.

**The PUSH Instruction**—Figure 6-26 illustrates how the PUSH instruction places data in the stack. The number  $01_{16}$  is in the accumulator and we wish to temporarily store it. While we could store it in memory, this would require a 2-byte or a 3-byte instruction. Instead, we use the PUSH instruction to place this number in the stack. Notice that the number is placed in the top location of the stack as shown in Figure 6-25A. The number remains there until we retrieve it or until we “push” another byte into the stack.

Figure 6-26B shows what happens if, at some later time, we “push” another byte into the stack. Notice that the accumulator now contains  $03_{16}$ . If the PUSH instruction is executed, the contents of the accumulator are pushed into the top of the stack. To make room for this new number, the original  $01_{16}$  is pushed deeper into the stack.

Figures 6-26C and 6-26D show two more numbers being pushed into the stack at later points in the program. Notice that the new data is always pushed into the top of the stack. To make room for the new data, the old data is pushed deeper into the stack. For this reason, this arrangement is often called a **push-down** or **cascade stack**. The name **cascade stack** comes from the characteristic cascading of data down through the stack as each new byte is pushed in at the top.

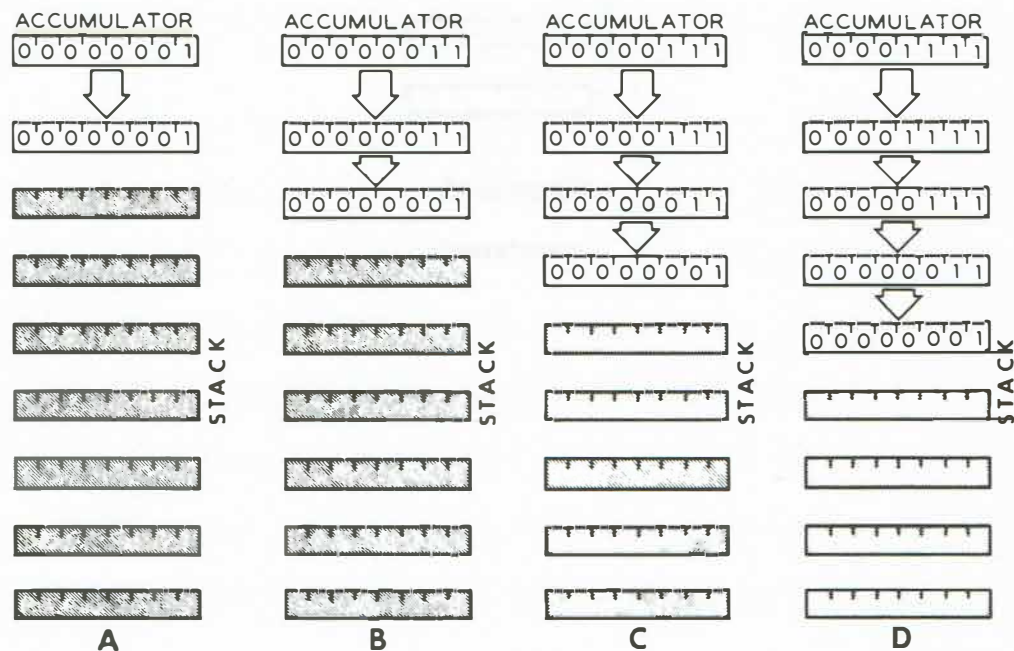


Figure 6-26

Pushing data into the stack.

**The PULL Instruction**—The MPU retrieves data from the stack by using the PULL instruction. In some microprocessors, this is referred to as a POP instruction.

Figure 6-27 illustrates how data can be pulled (or popped) from the stack. Figure 6-27A shows the stack as it appeared after the last push operation. Notice that it contains four bytes of data, with the last byte of data being at the top of the stack.

The PULL instruction retrieves the byte that is at the top of the stack. As this byte is removed from the stack, all other bytes move up, filling in the space left by that byte. Figure 6-27B illustrates how  $0F_{16}$  is pulled from the stack. Notice that  $07_{16}$  is now at the top of the stack.

Figures 6-27C and 6-27D show how the next two bytes can be pulled from the stack. In each case, the remaining bytes move up in the stack, filling in the register vacated by the removed byte.

If you compare Figures 6-26 and 6-27, you will notice that the data must be pulled from the stack in the reverse order from which it was pushed in. Hence, the last byte pushed into the stack is the first byte that is pulled from the stack. Another name for this arrangement is a last-in/first-out (LIFO) stack.

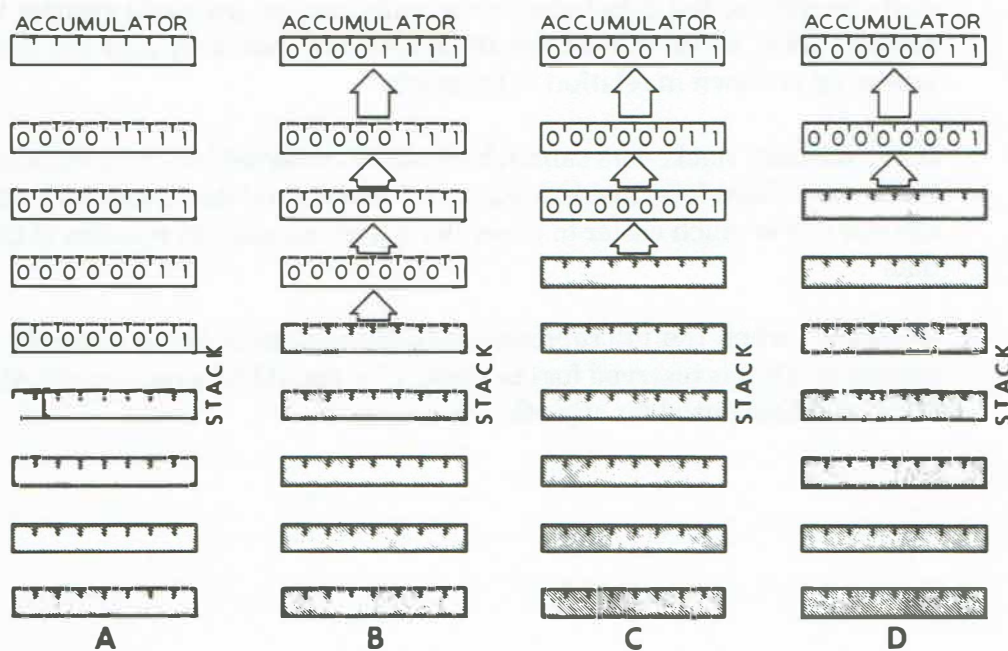


Figure 6-27

Pulling data from the stack.

## Memory Stack

While a cascade stack has its merits, it does have some limitations. For one thing, the number of registers is generally quite limited, with eight being typical. If more than eight pieces of data are pushed into the stack, the “older” bytes are pushed out the bottom and are lost. Also, the read-out of the stack is destructive. When a byte is pulled from the stack, it no longer exists in the stack. This is fundamentally different than reading a byte from memory.

Because of these limitations, the 6808 MPU does not use a cascade stack. Instead, a section of RAM can be set aside by the programmer to act as a stack. This has several advantages. First, the stack can be any length that the programmer requires. Second, the programmer can set up more than one stack if it is so desired. Third, the stack data can be addressed using any of the instructions that address memory.

**Stack Pointer**—Recall that the 6808 MPU has a 16-bit register called the stack pointer. In a memory-type stack, the stack pointer defines the memory location that acts as the top of the stack.

The cascade stack considered earlier generally does not require a stack pointer. The top of the stack is determined by hardware. During push and pull operations, the data bytes are actually moved from one register to another. That is, the top of the stack remains stationary and the data moves up or down in relation to the stack.

In the memory stack, data cannot be easily transferred from one location to the next. Therefore, instead of moving data up and down in relation to the stack, it is much easier to move the top of the stack in relation to the data.

Generally, when the microprocessor-based system is being planned, a section of RAM is reserved for the stack. This should be a section of RAM that is not being used for any other purpose.



Once this is done, the stack can be set up by a program. The top of the stack is established by loading an address into the stack pointer. For example, suppose we wish to establish address  $01F9_{16}$  as the top of the stack. The following instruction could be used:

```
LDS#
01
F9
```

This loads the address  $01F9_{16}$  into the stack pointer and establishes that address as the top of the stack. However, as you will see, the top of the stack moves each time data is pushed into—or pulled from—the stack.

**The PUSH Instructions**—The 6808 MPU has two push instructions, PSHA and PSHB. These single-byte instructions push the contents of their respective accumulator onto the stack.

Figure 6-28 shows the effects of the PSHA instruction. Before the instruction is executed, the stack pointer contains the address  $01F9_{16}$  as a result of a previous LDS instruction. Accumulator A contains a data byte ( $AA_{16}$ ). If the push instruction is now executed, the contents of accumulator A are pushed into memory location  $01F9_{16}$ . Then the stack pointer is automatically decremented to  $01F8_{16}$ . This automatically moves the top of the stack as shown.

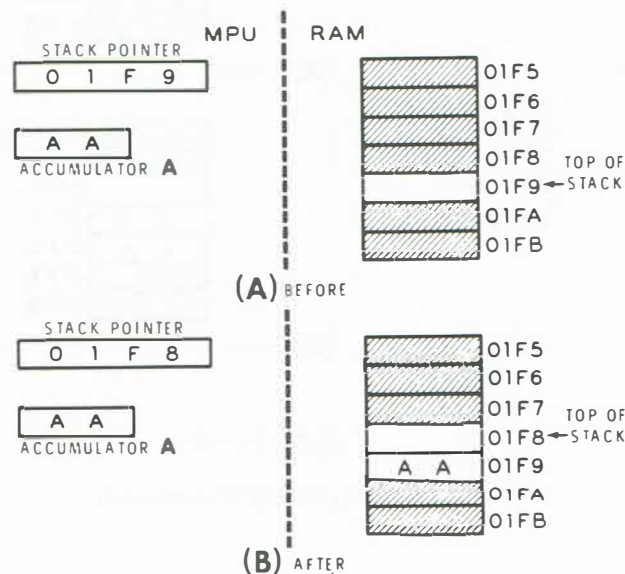


Figure 6-28

Executing the PSHA instruction.

If you look at your Instruction Set Summary card, you will see that the operation is described as follows.

$$A \rightarrow M_{SP}, SP - 1 \rightarrow SP$$

This means that the contents of the A accumulator are transferred to the memory location specified by the stack pointer. Also, the contents of the stack pointer are replaced by the previous contents of the stack pointer minus one. In other words, after the accumulator-to-stack transfer takes place, the stack pointer is decremented by one.

To reinforce the idea, assume that at some later point in the program, the MPU executes a PSHB instruction. This is illustrated in Figure 6-29. Before PSHB is executed, the B accumulator contains  $BB_{16}$  and the stack pointer is still pointing to  $01F8_{16}$ . When PSHB is executed, the contents of accumulator B are pushed onto the stack and the stack pointer is decremented to  $01F7_{16}$ .

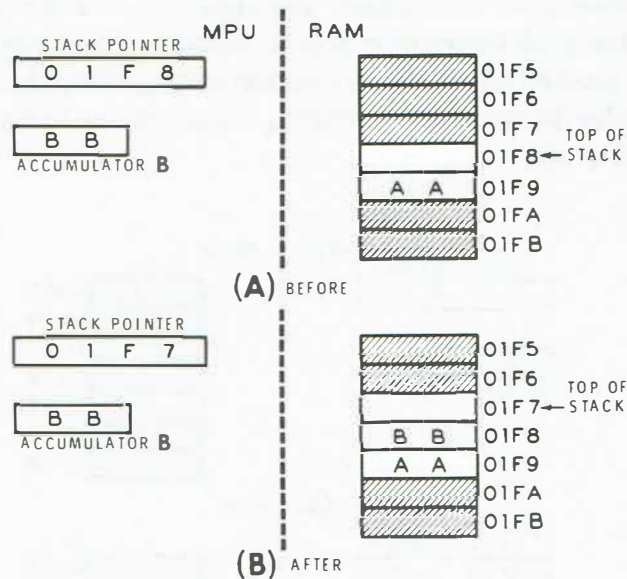


Figure 6-29

Executing the PSHB instruction.

**The PULL Instructions**—Data bytes are moved from the stack with the pull instruction. The 6808 MPU has two pull instructions. PULA allows the MPU to pull data from the stack into the A accumulator. PULB performs a similar operation except the data byte goes into accumulator B. In each case, data is pulled from the top of the stack. Thus, the data byte available to the MPU is the last byte that was placed in the stack.

For example, Figure 6-30A shows the stack as we left it after the last push instruction. Figure 6-30B shows what happens if the PULA instruction is executed. First, the stack pointer is automatically incremented by one to  $01F8_{16}$ . Then, the contents of the memory location designated by the stack pointer are transferred to accumulator A. Thus,  $BB_{16}$  goes into accumulator A. Notice that the stack pointer is incremented **before** the byte is pulled from the stack.

To be certain you have the idea, consider what happens if the PULB instruction is now executed. Figure 6-30C shows that the stack pointer is automatically incremented to  $01F9_{16}$ . The contents of that location are then pulled into accumulator B. This operation is described on your Instruction Set Summary card as:

$$SP + 1 \rightarrow SP, M_{sp} \rightarrow B.$$

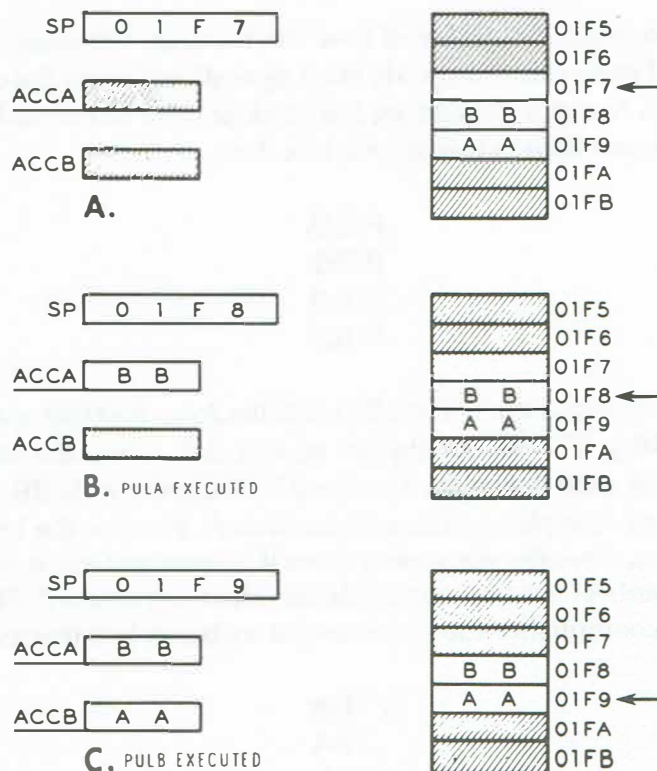


Figure 6-30

Executing the PULL instructions.

**Using the Stack**—Figure 6-31 summarizes all of the instructions that differently affect the stack. Find these instructions on your Instruction Set Summary card. The push and pull instructions are listed with the Accumulator and Memory Operations. Those instructions that affect the stack pointer are listed under Index Register and Stack Pointer Operations.

ADDRESSING MODES

STACK AND STACK POINTER		ADDRESSING MODES												BOOLEAN/ARITHMETIC OPERATION			
OPERATIONS	MNEMONIC	IMMED			DIRECT			INDEX			EXTND			INNER			(All register labels refer to contents)
		OP	~	#	OP	~	#	OP	~	=	OP	~	#	OP	~	#	
Push Data	PSHA													36	4	1	$A \rightarrow M_{SP}, SP - 1 \rightarrow SP$
	PSHB													37	4	1	$B \rightarrow M_{SP}, SP - 1 \rightarrow SP$
Pull Data	PULA													32	4	1	$SP + 1 \rightarrow SP, M_{SP} \rightarrow A$
	PULB													33	4	1	$SP + 1 \rightarrow SP, M_{SP} \rightarrow B$
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$
Load stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_{16}, (M - 1) \rightarrow SP_{16}$
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_{16} \rightarrow M, SP_{16} \rightarrow (M - 1)$
Indx Reg $\rightarrow$ Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$
Stack Pntr $\rightarrow$ Indx Reg	TSX													30	4	1	$SP - 1 \rightarrow X$

Figure 6-31

Stack and stack pointer instructions.

Following are some examples of how the stack can be used. First, consider a trivial example. Using only stack operations, swap the contents of accumulators A and B. Assuming the stack pointer has already been set up, the program segment might look like this:

```
PSHA
PSHB
PULA
PULB
```

Assume that accumulator A initially contains  $AA_{16}$  and that accumulator B contains  $BB_{16}$ . The first operation pushes  $AA_{16}$  onto the stack. Next,  $BB_{16}$  is pushed onto the stack. The third instruction pulls  $BB_{16}$  from the top of the stack and places it in accumulator A. Finally, the last instruction pulls  $AA_{16}$  from the stack and places it in accumulator B. As you can see, the contents of the two accumulators are now reversed. The following routine accomplishes the same results with one less instruction:

```
PSHA
TBA
PULB
```

Now, look at a more complex example. Assume that you wish to transfer  $16_{10}$  bytes of data from one place in memory to another. As you saw earlier, this type of problem is a good candidate for indexing. However, indexing alone becomes cumbersome if the two lists are over  $FF_{16}$  memory locations apart. The reason for this is that the offset address can only extend  $FF_{16}$  locations above the address in the index register.

In this example, assume you wish to move the data in memory locations  $0010_{16}$  through  $001F_{16}$  to locations  $01F0_{16}$  to  $01FF_{16}$ . While you could do this using indexing alone, the program becomes unnecessarily complicated. Two separate indexes must be maintained; one for loading data from  $0010_{16}$  through  $001F_{16}$ , the other for storing data in  $01F0_{16}$  through  $01FF_{16}$ . A simpler approach is to use indexing for one operation and the stack capability for the other operation. That is, we could load data from the lower list using indexing and store it in the upper list using the stack capability.

A program that does this is shown in Figure 6-32. The first instruction loads the stack pointer with address  $01FF_{16}$ . This is the address of the last entry in the new list that will be formed. Recall that the new list is to be written in locations  $01F0_{16}$  through  $01FF_{16}$ . Once location  $01FF_{16}$  is established at the top of the stack, we can enter data into the new list simply by pushing data onto the stack. Because the stack pointer is decremented with each push operation, we must push the last entry in the list onto the stack first.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0020	8E	LDS#	Load the stack pointer immediately with the
0021	01	01	address of the last entry in the
0022	FF	FF	new list.
0023	CE	LDX#	Load the index register immediately with the
0024	00	00	address of the last entry in the
0025	1F	1F	original list.
0026	A6	LDAA X	Load accumulator A indexed from
0027	00	00	the original list.
0028	36	PSHA	Push the contents of accumulator A into the new list.
0029	09	DEX	Decrement the index register.
002A	8C	CPX#	Compare the contents of the index register
002B	00	00	with one less than the address of the
002C	0F	0F	first entry in the original list.
002D	26	BNE	If no match occurs, branch back
002E	F7	F7	this far.
002F	3E	WAI	Otherwise, wait.

Figure 6-32

Moving a list of data using both indexing and stack operations.



The second instruction loads the index register with the address of the last entry in the original list. This is necessary for the reason pointed out above.

Next, the A accumulator is loaded using indexed addressing. Since the offset address is  $00_{16}$ , the accumulator is loaded with the contents of  $001F_{16}$ . That is, the last entry in the original list is loaded into accumulator A.

The PSHA instruction then pushes the contents of accumulator A onto the stack. Thus, the last entry in the original list is transferred to location  $01FF_{16}$ . In the process, the stack pointer is automatically decremented to  $01FE_{16}$ .

The index register is decremented to  $001E_{16}$  by the next instruction. Then, the CPX instruction compares the index register with  $000F_{16}$  to see if all entries in the list have been moved. If no match occurs, the MPU branches back and picks up the next entry in the list. The loop is repeated over and over again until the entire list has been moved to its new location.

Other uses of the stack will be discussed later. However, even if the stack did nothing more than has already been explained, it would be very useful to have. But as you will see, the MPU uses the stack in several other ways that makes it even more important.



## Programmed Review

35.	A _____ is a group of temporary locations in which data can be stored and later retrieved.
36.	(stack) The 6808 MPU uses a _____ stack. (cascade/memory)
37.	(memory) In a microprocessor using a cascade type stack, all data transfers are between the _____ of the stack and the accumulator. (top/bottom)
38.	(top) The _____ instruction stores data in the stack.
39.	(PUSH) The _____ instruction retrieves data stored in the stack.
40.	(PULL) The _____ indicates the address at the top of the stack.
41.	(stack pointer) The _____ instruction transfers data from the top of the stack to accumulator B.
	(PULB)

## SUBROUTINES

A subroutine is a group of instructions that perform some limited but frequently required task. A given subroutine may be used many times during the execution of the main program. In many cases, the easiest way to write a program is to break the overall job down into many simple operations, each of which can be performed by a subroutine.

Because subroutines are used so frequently, most microprocessors have special capabilities that allow them to handle subroutines efficiently. In this section, these capabilities will be examined. The discussion will start with the instructions associated with subroutines.

The 6808 MPU has three instructions that are used to handle subroutines. They are:

Jump to Subroutine (JSR).  
Branch to Subroutine (BSR).  
Return from Subroutine (RTS).

Each of these will be discussed in this section. One other instruction that has not yet been mentioned will also be discussed. It is the Jump (JMP) instruction. While not used exclusively with subroutines, the JMP instruction makes an excellent introduction to the Jump-to-Subroutine (JSR) instruction. Therefore, the (JMP) instruction will be discussed first.

### Jump (JMP) Instruction

This instruction allows the MPU to jump from one point in a program to another. In this respect, it is somewhat like the Branch-Always (BRA) instruction that was discussed earlier. The difference is the method of addressing used. Recall that the BRA instruction used relative addressing which has the advantage that only a 2-byte instruction is required. Its disadvantage is that the branch must be within the range of  $-128_{10}$  bytes to  $+127_{10}$  bytes of the program count.

The JMP instruction can use either the indexed or the extended addressing mode. It does not use relative addressing. When using extended addressing, the format of the JMP instruction is as shown in Figure 6-33. Three bytes are required; the opcode, followed by the 2-byte address to which the MPU is to jump. Since a 16-bit address is given, the jump may be to any point in the  $65,536_{10}$  byte memory range. This address is loaded into the program counter so that the next opcode is fetched from that address. The previous contents of the program counter are lost. Thus, the MPU starts executing instructions from a new point in memory.

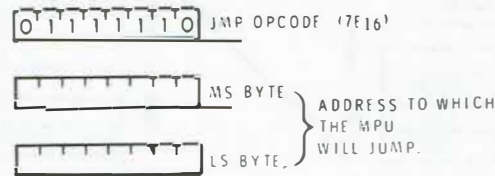


Figure 6-33

Format of the JMP instruction using extended addressing.

As an example of how the JMP instruction can be used is shown in Figure 6-34. Here, a long program is to be repeated over and over. This is typical of applications such as robots that repeat the same operations endlessly. The program is contained in the upper 1k bytes of memory. It starts at location  $FC00_{16}$  and ends at  $FFE0_{16}$ . Notice that the last instruction is JMP  $FC00_{16}$ . This sends the program back to its beginning so that the loop is repeated endlessly.

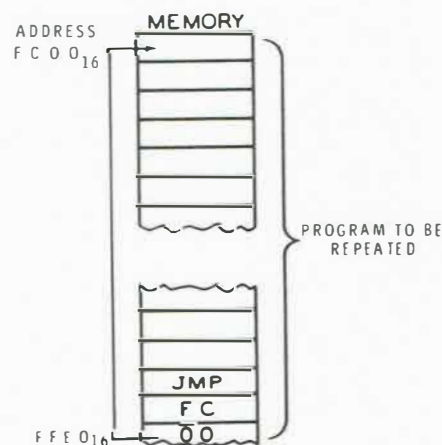


Figure 6-34

Using the JMP instruction to repeat a program.

Another possible use of the JMP instruction is shown in Figure 6-35. Here, the main program is in the lower memory locations shown on the left. The main program requires a subroutine that is at address A000 (shown on the right). The JMP instruction at address 0070 sends the MPU off to the subroutine as shown. The last instruction in the subroutine is another JMP instruction that sends the MPU back to the main program.

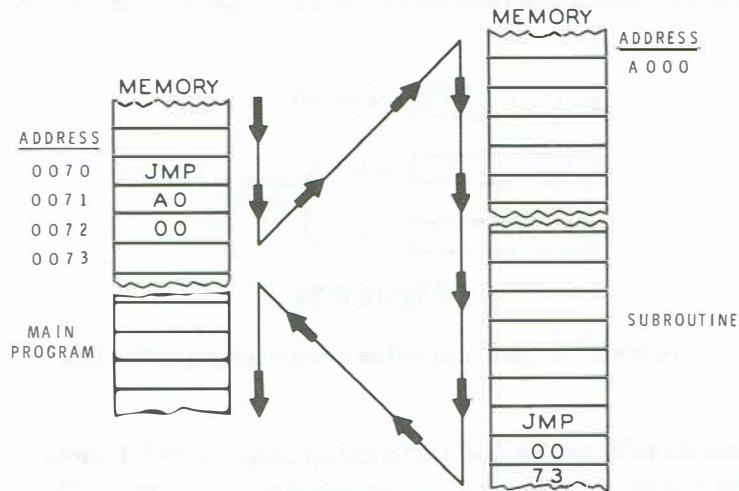


Figure 6-35

### Using the JMP instruction to call a subroutine.



Jumping to a subroutine is often referred to as **calling** a subroutine. While we can call a subroutine using the JMP instruction, this approach has a distinct problem. What happens if the main program wants to call the same subroutine more than once? That is, suppose a situation like that shown in Figure 6-36 is required. Here, the main program (on the left) wishes to call the subroutine (on the right) at two separate points. Jumping to the subroutine presents no problem; we can do that as many times as we desire, using the instruction JMP A000. The problem is: how do we get back from the subroutine to the main program? The first time through the subroutine, the MPU should return to address 0073. The second time through, the MPU should return to address 0093.

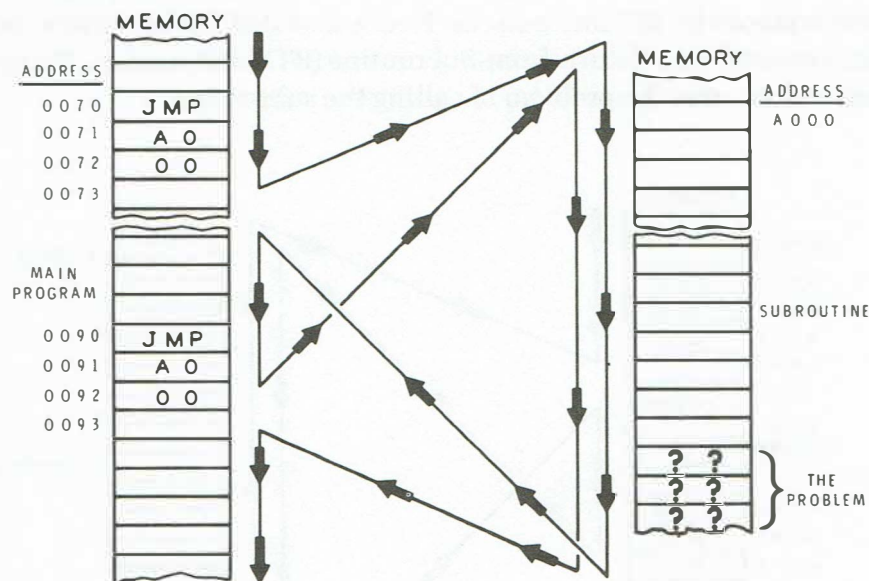


Figure 6-36

The JMP instruction cannot handle situations like this one.

A programmer could get around this problem by changing the last instruction in the subroutine before each call or by constructing a table of return addresses, etc. However, most microprocessors have some instructions that solve this problem for us. The following section will discuss the 6808 MPU's solution to this problem.

## JSR and RTS Instructions

If you refer to Figure 6-36 again, you will see that this problem arises because the old program count is not saved when the MPU jumps from one location to the next. However, the 6808 MPU has an instruction that will not only jump to a subroutine, it will also cause the old program count to be stored away. This instruction is called the Jump-to-Subroutine (JSR) instruction. Its format is exactly the same as the JMP instruction, but its execution is different.

Figure 6-37 shows how the earlier problem can be solved using the JSR instruction. Notice that the two JMP instructions in the main program have been replaced by JSR instructions. Notice also that the last instruction in the subroutine is a Return-from-Subroutine (RTS) instruction. These new instructions ease the problem of calling the subroutine.

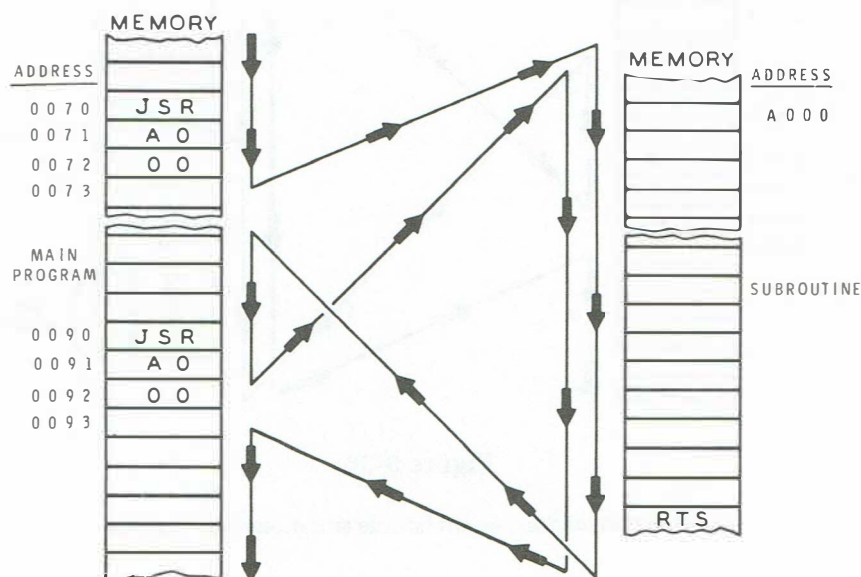


Figure 6-37

The JSR and RTS instructions can be used to handle this situation.

When the first JSR instruction is executed, the subroutine address  $A000_{16}$  is placed in the program counter. However, just prior to this, the program counter was incremented to the address of the next instruction in sequence. That is, the program counter was advanced to  $0073_{16}$  while the contents of address  $0072_{16}$  were being retrieved. This count ( $0073_{16}$ ) is automatically pushed onto the stack. By saving the old program count, the MPU can tell where to return after the subroutine is finished. As soon as the old program count is tucked away safely in the stack, the subroutine address  $A000_{16}$  is placed in the program counter. Thus, the MPU fetches the next instruction from address  $A000_{16}$ .

Notice that the last instruction in the subroutine is an RTS instruction. When the MPU encounters this single-byte instruction, it will jump back to the point where it left off in the main program. It does this by pulling the old program count ( $0073_{16}$ ) from the stack and placing it in the program counter. Consequently, the next instruction will be fetched from address  $0073_{16}$ . As you can see, this returns the MPU to the correct point in the main program.

Notice that the programmer does not specify a return address at the end of the subroutine. Instead, the return address is automatically pulled from the stack. This allows us to call the subroutine repeatedly from several different points in the main program.

Figure 6-37 shows that the subroutine is called again by the JSR  $A000$  instruction in location  $0090_{16}$ . As this instruction and address are decoded, the program count is incremented to  $0093_{16}$ . This program count is pushed onto the stack. Then  $A000_{16}$  is placed in the program counter. Thus, the MPU jumps off to the subroutine. When the subroutine is finished, the RTS instruction causes the old program count to be pulled from the stack into the program counter. This causes the MPU to jump back to address  $0093_{16}$  which contains the next instruction in the main program.

## Nested Subroutines

Figure 6-38 shows a situation in which the main program calls a subroutine A. In turn, subroutine A calls subroutine B. In this situation, subroutine B is called a “nested” subroutine. That is, a nested subroutine is a program segment that is called by another subroutine. If control is to be eventually returned to the main program, two program counts must be saved. Figure 6-38 shows how the two program counts are saved in the stack.

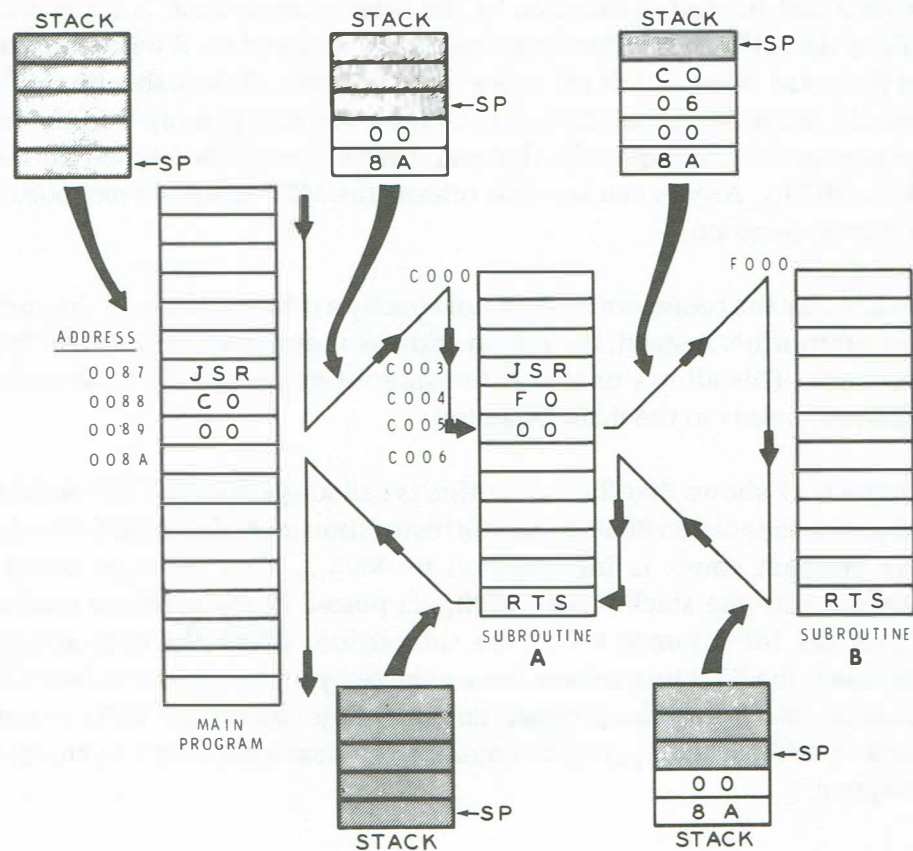


Figure 6-38

Handling “nested” subroutines.

At the start of the main program, the stack pointer is loaded with the address of the area in memory that has been set aside to act as the stack. If no stack instructions have been executed when the main program arrives at the first JSR instruction, the stack pointer will still be pointing to where it was originally set. The contents of the stack are of no interest until this point.

When the main program reaches the JSR instruction, the program count is advanced to the address of the next instruction in sequence ( $008A_{16}$ ). When the JSR instruction is executed, this address ( $008A_{16}$ ) is pushed onto the stack as shown. The low-order byte goes in first, followed by the high-order byte. In the process, the stack pointer is decremented twice. Finally, the new address ( $C000_{16}$ ) is placed in the program counter. This causes the MPU to jump off to subroutine A, which starts at  $C000_{16}$ .

Notice that halfway through subroutine A, subroutine B is called. Consequently, the return address in subroutine A ( $C006_{16}$ ) must be saved. That is, when the program reaches the JSR instruction in subroutine A, the return address ( $C006_{16}$ ) is pushed onto the stack as shown. Notice that there are now two return addresses in the stack. The starting address of subroutine B ( $F000_{16}$ ) is then placed in the program counter and the MPU jumps off to this subroutine.

Subroutine B has no nested subroutines of its own, so the program flow is through the subroutine as shown. The last instruction in subroutine B is the RTS instruction. At this point, the MPU pulls the return address ( $C006_{16}$ ) from the top of the stack and places it in the program counter. This causes the MPU to jump back to the instruction at address  $C006_{16}$  in subroutine A.

The remainder of subroutine A is then executed down to the RTS instruction. This instruction causes the MPU to pull the next address ( $008A_{16}$ ) from the stack and place it in the program counter. Notice that this sends the MPU back to the main program.

For simplicity, a single level of subroutine nesting is shown in this example. However, in practice, many levels of nesting may be used. For example, subroutine B could call subroutine C; etc. Any level of nesting can be used as long as enough memory is set aside for the stack. Remember, each return address requires two bytes in the stack.



## Branch to Subroutine (BSR) Instruction

Quite often, the subroutine we wish to call is within the  $-128_{10}$  to  $+127_{10}$  byte range of the relative address. When it is, we can save one byte by using the Branch-to-Subroutine (BSR) instruction. The execution of BSR is identical to that of JSR except that relative addressing is used. The old program count is saved in the stack before the branch occurs. Thus, the RTS instruction at the end of the subroutine will cause the old program count to be restored.

## Summary of Subroutine Instructions

Figure 6-39 shows the four instructions discussed in this section. Notice that the BSR instruction uses relative addressing. The JMP and JSR instructions can use either indexed or extended addressing. The RTS instruction uses inherent addressing, since its address is pulled from the top of the stack.

JUMP AND BRANCH OPERATIONS	MNEMONIC	RELATIVE			INDEX			EXTND			INHER		
		OP	#		OP	#		OP	#		OP	#	
Branch To Subroutine	BSR	8D	8	2									
Jump	JMP				6E	4	2	7E	3	3			
Jump To Subroutine	JSR				AD	8	2	8D	9	3			
Return From Subroutine	RTS										39	5	1

Figure 6-39

Subroutine and jump instructions.

Find these instructions on your Instruction Set Summary card. The operations performed by these instructions are illustrated under "Special Operations" on the back of the card. Also, **Appendix C** of this course gives a concise description of the operations performed by each of these instructions.

## Programmed Review

42. A \_\_\_\_\_ is a group of instructions that perform some specific, limited task that is used more than once by the main program.

43. (subroutine) The BRA instruction uses \_\_\_\_\_ addressing; therefore, it can only be used in the  $-128_{10}$  to  $+127_{10}$  byte range.

44. (relative) The JMP instruction can use either \_\_\_\_\_ or \_\_\_\_\_ addressing.

45. (extended, indexed) Jumping to a subroutine is often referred to as \_\_\_\_\_ a subroutine.

46. (calling) One difference between the JMP and JSR instruction is that when the \_\_\_\_\_ instruction is executed the program count is saved.

47. (JSR) When the program count is saved, it is pushed into the top \_\_\_\_\_ locations of the stack.

48. (two) Generally, the last instruction in the subroutine will be the \_\_\_\_\_ instruction.

49. (RTS) When subroutine A calls subroutine B, subroutine B is said to be \_\_\_\_\_.

50. (nested) The RTS instruction uses \_\_\_\_\_ addressing.

(inherent)

## INPUT—OUTPUT (I/O) OPERATIONS

A full explanation of input-output (I/O) operations will be given in Unit Ten, but a brief introduction to I/O is necessary at this point. In this section, you will learn what is involved in sending data to—or taking data from—the MPU.

To be useful, a microprocessor system, especially a microprocessor-controlled robot, must accept data from the outside world, process it in some way, and present results to the outside world. The input device may be nothing more than a group of switches while the output device can be as simple as a bank of indicator lamps. On the other hand, a single microprocessor-controlled robot might have several inputs from complex sensing devices. The point is, I/O requirements can vary greatly from one application to the next. In this section, we will be concerned with the simplest form of I/O operations.

In the short history of microprocessors, two distinctly different methods have been developed for handling I/O operations. In some microprocessors, I/O operations are handled by I/O instructions. These microprocessors generally have one **input** instruction and one **output** instruction. When the input instruction is executed, a byte is transferred from the selected I/O device to a register (usually one of the accumulators) in the MPU. The I/O device is selected by sending out a device selection byte on the address bus. By using an 8-bit byte for device selection, the MPU can specify  $256_{10}$  different I/O devices. Of course, no microprocessor system uses that many devices, but the capability is there.

The output instruction causes a data transfer from the accumulator to the selected I/O device. While this method of handling I/O operations is used in many microprocessors, the 6808 MPU uses a different technique.

The other method for handling I/O operations is to treat all I/O transfers as memory transfers. This is the method used by the 6808 MPU and many other microprocessors. In fact, even those microprocessors that have I/O instructions can ignore those instructions and handle I/O operations as memory transfers.

The 6808 MPU has no I/O instructions. Instead, an I/O device is assigned an address and is treated as a memory location. For example, assume that an input keyboard has been assigned an address of  $8000_{16}$ . We can input data into accumulator A by using the instruction:

LDAA  $8000_{16}$ .

By the same token, an output display may have been assigned the address  $9000_{16}$ . In this case, we can output from accumulator B using the instruction:

STAB  $9000_{16}$ .

As you can see, the I/O device is treated as a memory location. The system block diagram shown in Figure 6-40 shows how an I/O device is connected to the microcomputer. Notice that both the data bus and the address bus connect to the I/O interface. As you will see in a later unit, the interface can consist of an address decoder, an output or input latch, and buffers or drivers.

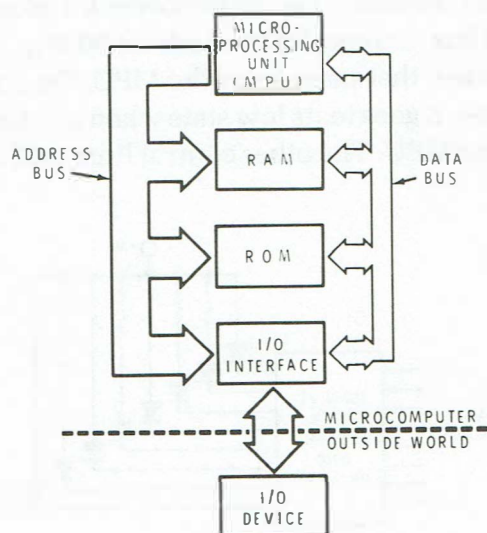


Figure 6-40

Adding I/O to the microcomputer.

The address decoder monitors the address bus and enables the interface circuitry whenever the proper address is detected. This prevents the I/O interface from interfering when data is being transferred between memory and the MPU.

The I/O interface will generally have an output latch if it is to be used for an output operation. The reason for this is that the data from the MPU will appear on the data lines for only an instant (usually less than one microsecond). By storing the output data in a latch, the I/O device is given a much longer period of time to examine and respond to the data.

Buffers or drivers are also included in the I/O interface. As you will see later, these are frequently necessary when several different circuits are sharing the same bus.

## Output Operations

Figure 6-41 shows a simplified output circuit. Here, the output device is a bank of eight light-emitting diodes (LEDs). Enough detail is shown to illustrate how an output operation can be performed. The address decoder monitors the address bus, looking for the address  $9000_{16}$ . It also monitors some of the control lines that connect to the MPU. One of those lines is called a read-write line. It goes to its low state when a write (output) operation is initiated by the MPU. The other control lines will be discussed in Unit Ten.

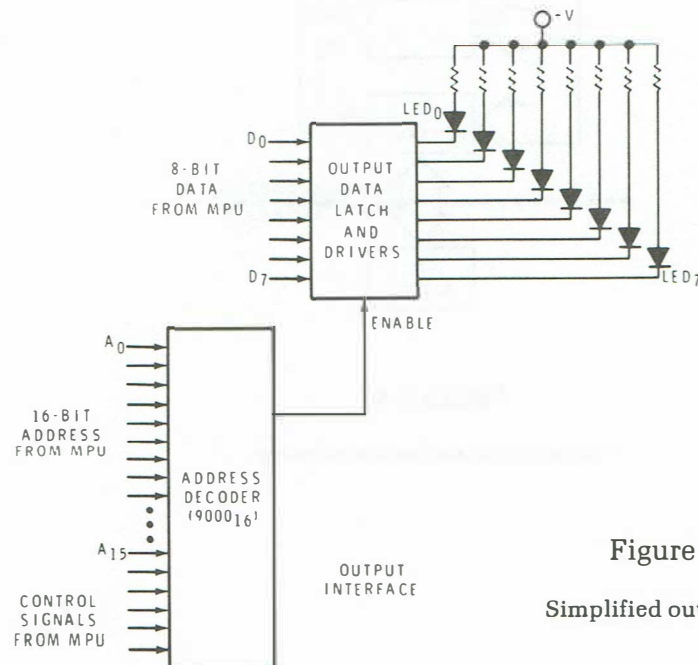


Figure 6-41  
Simplified output circuit.



Notice that the output of the address decoder is used to enable the output data latches and drivers. When these are enabled, the byte on the data lines is stored in the latch. The data bits stored in the latch cause the appropriate LEDs to light up. By outputting appropriate bit patterns, the MPU can cause different binary numbers to be displayed.

Notice that the address decoder (and therefore the display) is given the address  $9000_{16}$ . We can output data to the display in several different ways. For example, we can load the appropriate pattern to be displayed into accumulator A. Then, by executing a "Store-Accumulator-A" extended instruction, we can transfer the contents of the accumulator to the display. The instruction would be:

STAA  $9000_{16}$ .

Or, we could output data from accumulator B by using the instruction:

STAB  $9000_{16}$ .

In either case, the address  $9000_{16}$  goes out on the address bus for a brief interval of time. The address decoder recognizes this address, and at the same time, the control lines indicate that an output operation is called for. In particular, the read-write line goes low. This causes the address decoder to enable the output data latch for an instant. Simultaneously, the 8-bit data byte appears on the data bus. The output latch stores the data byte. The data appears at the input of the latch for less than a microsecond (typically). However, once the data is stored, it appears at the output of the latch until new data is written in. Thus, the output data will be displayed until the next byte of data is outputted by the MPU.

## Input Operations

Figure 6-42 shows a simplified input circuit in which the input device is a bank of eight switches. When a switch is open, its respective input line to the buffer is held high by the pull-up resistors,  $R_0$  through  $R_7$ . However, when a switch is closed, its respective input line is pulled low because the switch connects it to ground.

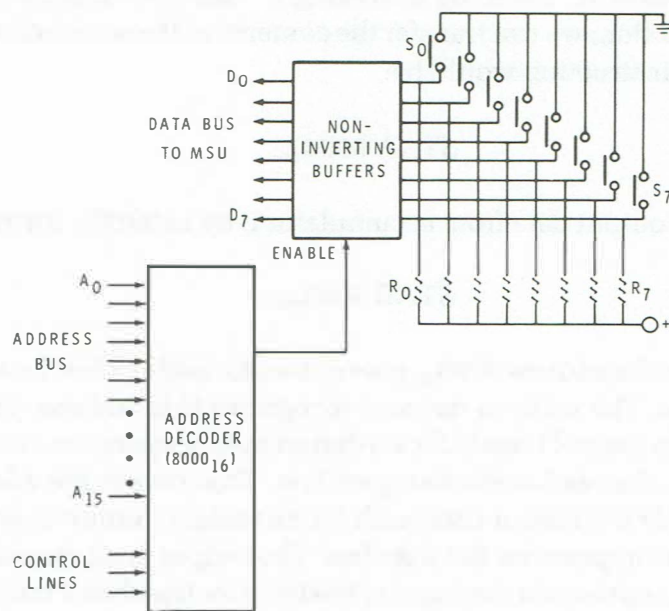


Figure 6-42

Simplified input circuit.

In this simple circuit, no latch is required between the switches and the data bus. However, a buffer is used so that the switch bank can be effectively disconnected from the data bus when the switches are not being addressed.

As with the output circuit, an address decoder monitors the address and control lines. Notice that the assigned address is  $8000_{16}$ . To input data from the switch bank to accumulator A, we use the instruction:

LDAA  $8000_{16}$ .

Or, we could input the data to accumulator B by using the instruction:

LDAB  $8000_{16}$ .

In either case, the address  $8000_{16}$  is placed on the address line. The address decoder recognizes this address and enables the buffer. For a brief interval, typically less than one microsecond, the lines of the data bus assume the same state as the lines on the right side of the buffer. If no switch is depressed, all data lines will be high and all 1's ( $FF_{16}$ ) will be loaded into the accumulator. However, if one of the switches ( $S_0$ , for example) is depressed, its respective data line (in this case  $D_0$ ) will be low. In this example, the number read into the accumulator will be  $FE_{16}$ . By examining the byte that is read in, the MPU can determine which switch is depressed.

## Input—Output Programming

You now know enough about simple input/output circuits to perform some I/O operations. Refer to Figure 6-41 and 6-42 for the first example. Now, assume that you would like one of the LEDs to light when the corresponding switch is pushed. That is, LED0 should light when  $S_0$  is pushed; LED1 should light when  $S_1$  is pushed, etc.

If you refer to Figure 6-41, you will see that placing a 0 in the proper bit in the latch causes an LED to light. For example, a 0 in bit 0 will cause LED0 to be forward biased, which will allow the diode to conduct, thus emitting light. Notice that a 1 at bit 0 will not allow the diode to conduct and emit light. Hence, a 0 turns the LED on and a 1 turns it off.

Referring to Figure 6-42, you will find that, when one of the switches is closed, its corresponding line goes to 0. If the switch is not closed, its corresponding line is a 1.

If we load data into one of the accumulators from address  $8000_{16}$  and then store the data at address  $9000_{16}$ , the switches will appear to control the LEDs. The program could look like this:

```
LDAA
80
00
STAA
90
00
BRA
F8
```

If S0, and only S0, is closed when the LDAA 8000 instruction is executed,  $1111110_2$  will be loaded into accumulator A. The next instruction stores this data byte in the output latch. This causes LED 0, and only LED 0, to light. The BRA instruction holds the MPU in a tight loop. Try a few examples and verify that each time a switch is closed, the corresponding LED will light. If the switches are set to some 8-bit binary number, the LEDs will display that 8-bit binary number.

Now, suppose we change our mind and decide that the LEDs should display the one's complement of the binary number set on the switches. We do not have to touch the hardware. Instead, we just change the program. The new program might look like this:

```
LDAA
80
00
COMA
STAA
90
00
BRA
F7
```

Notice that we have simply inserted the one's complement instruction between input and output operations.

As another example, suppose we wish to display a number that is four times greater than the number set on the switches. Our program could be modified to this:

```
LDAA
80
00
ASLA
ASLA
STAA
90
00
BRA
F6
```

Once again, no hardware change is needed; we simply insert two ASLA instructions between the input and output operations. Recall that each ASL instruction, in effect, doubles the contents of the associated accumulator. Thus, two ASLA instructions will quadruple the contents of accumulator A.

Although these examples are very simple, they illustrate the flexibility of this I/O arrangement. Data is pulled from the input device as if it were being pulled from memory. The data can then be transferred to the output devices as if it were being stored in memory. While the data is in the MPU, it can be modified in any number of ways. The input byte can be shifted left or right, it can be added to or subtracted from another number, and it can be ANDed or ORed with another byte. The possibilities are endless, and yet, none of these involve a hardware change. All data manipulations can be accomplished by the program.



## Program Control of I/O Operations

In the preceding examples, all I/O transfers are controlled by the program and the program alone. The program is in a tight loop that inputs data from the switches, modifies the data (if required), and outputs the data to the displays.

When this arrangement is used, the MPU never knows if the data at the input is changed. It simply reads in the data a set number of times each second. By the same token, the MPU outputs the data over and over. This system works well for simple I/O operations; however, as the I/O requirements become more complex, this technique becomes cumbersome.

The program must be in a loop if it is to repeatedly check for inputs and refresh the output. As the number of data manipulations increase, the loop becomes longer and the MPU must check the inputs less frequently. When several I/O devices are used, as in the case of a microprocessor-controlled robot, it must check each input and refresh each output repeatedly. However, if the loop becomes too long, the MPU may miss a momentary switch closure. This may be acceptable in some applications, but in many others, it may be intolerable. Obviously then, a more sophisticated method of handling I/O operations must be available to the micro-computer.

## Interrupt Control of I/O Operations

A more effective way of handling I/O operations involves a concept called interrupts. Interrupts are a means by which an I/O device can notify the MPU that it is ready to send input data or to accept output data. Generally, when an interrupt occurs, the MPU suspends its current operation and takes care of the interrupt. That is, it might read-in or write-out a byte of data. After it has taken care of the interrupt, the MPU returns to its original task and takes up where it left off.

The following analogy may help you visualize an interrupt operation. Compare the MPU to the president of a corporation who is writing a report. The interrupt can be compared to a telephone call. The president's main task is the report. However, if the telephone rings, causing an interrupt, she finishes writing the present word or sentence, then answers the phone call. After she has attended to the phone call, she returns to the report and takes up where she left off. In this analogy, the ringing of the telephone notifies the president of the interrupt request.

This analogy shows the difficulty of the program-controlled I/O technique discussed earlier. If we remove the interrupt request (the ringing of the phone), we are left with an almost comical situation. The president writes a few words of the report, and then picks up the phone to see if anyone is on the other end. If not, she hangs up the phone, writes a few more words, and checks the phone again. Clearly, this technique wastes an important resource—the president's time.

This simple analogy shows the importance of an interrupt capability. Without it, a great deal of the MPU's time can be wasted doing routine operations. The last section of this unit will examine the interrupt capabilities of the 6808 MPU.

## Programmed Review

51. The microprocessor system communicates with the outside world through \_\_\_\_\_ devices.

52. (input/output or I/O) A group of switches could be a simple \_\_\_\_\_ device while a bank of LEDs could be a simple \_\_\_\_\_ device.

53. (input, output) In the 6808 MPU, all I/O transfers are treated as \_\_\_\_\_ transfers.

54. (memory) In the 6808 MPU, the \_\_\_\_\_ instruction can be used for transferring data from an I/O device to accumulator A.

55. (LDAA) In the 6808 MPU, the \_\_\_\_\_ instruction can be used for transferring data from accumulator B to an I/O device.

56. (STAB) The most effective way of handling I/O operations is through \_\_\_\_\_ control.  
(interrupt/program)

(interrupt)

## INTERRUPTS

Interrupts were briefly discussed in the previous section in connection with I/O operations. While I/O operations use part of the interrupt capability of the MPU, interrupts are also used in other ways. The 6808 MPU has four different types of interrupts:

Reset.

Non-Maskable Interrupt (NMI).

Interrupt Request (IRQ).

Software Interrupt (SWI).

This last section will examine each of these interrupts in detail.

## Reset

In a typical application, the microcomputer has a control or monitor program in a read-only-memory (ROM). Also, a random read-write memory (RAM) is used for holding input data, intermediate answers, output data, etc. As we have seen, the 6808 MPU can address up to  $65,536_{10}$  memory locations. Most microprocessor applications do not require this much memory. In many applications, the control program requires less than ten percent of the possible locations. The RAM probably uses less than two percent. Generally, the monitor program is placed at the high memory addresses. The RAM is usually given the low memory addresses so that the direct addressing mode can be used. The I/O devices are given intermediate addresses. Thus, the memory addresses may be allocated as shown in Figure 6-43.

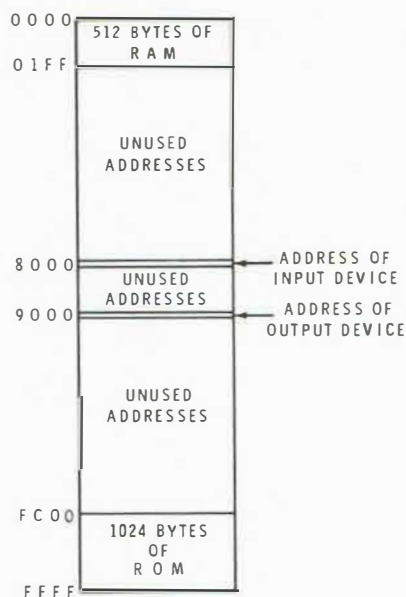


Figure 6-43

Memory allocations in a typical microcomputer system.

Notice that the control or monitor program is placed in a ROM at the very top of memory. In this example, a  $1024_{10}$  byte ROM is used. The addresses of the ROM are  $FC00_{16}$  through  $FFFF_{16}$ . A small RAM is placed at the low end of memory—addresses  $0000_{16}$  through  $01FF_{16}$  are used. Notice that all other addresses are unused except for two. The input device is assigned address  $8000_{16}$  while the output device is assigned address  $9000_{16}$ .



The monitor program stored in the ROM controls all the activities of the MPU. At all times, the entire system is being run by this program. In this example, when the microprocessor is initially turned on, it should start executing instructions at address  $FC00_{16}$ . Also, we should be able to restart the program at this address at any time. In order to accomplish this, the 6808 MPU has a built-in reset capability.

The 6808 MPU has a signal line or control pin that is called  $\overline{\text{reset}}$ . This pin or line is connected to a reset switch of some kind. If this line goes low for a prescribed period of time and then swings high, the MPU will initiate a **reset interrupt sequence**. The main purpose of the reset interrupt sequence is to load the address of the first instruction to be executed into the program counter. This would be easy to accomplish if, in every application, the starting address were the same. However, the starting address differs from one application to the next. Therefore, a convenient means is provided to allow the designer to specify any starting address that he likes.

In any 6808-based microprocessor system, the upper eight bytes of ROM are reserved for **interrupt vectors**. An interrupt vector is simply an address that is loaded into the program counter when an interrupt occurs. Figure 6-44 shows how these eight reserved memory bytes are allocated. Notice that addresses  $FFFE_{16}$  and  $FFFF_{16}$  contain the reset vector. That is, these two memory locations contain the address of the first instruction that is to be executed when the microcomputer is initially started. In our example, the first instruction in the monitor program is at address  $FC00_{16}$ . Consequently, this is our reset vector. Location  $FFFE_{16}$  must contain the high byte of the address ( $FC_{16}$ ) and  $FFFF_{16}$  must contain the low byte of the address ( $00_{16}$ ).

Address	
F F F 8	Interrupt Request Vector (high order address)
F F F 9	Interrupt Request Vector (low order address)
F F F A	Software Interrupt Vector (high order address)
F F F B	Software Interrupt Vector (low order address)
F F F C	Non-Maskable-Interrupt Vector (high order address)
F F F D	Non-Maskable-Interrupt Vector (low order address)
F F F E	Reset Vector (high order address)
F F F F	Reset Vector (low order address)

Figure 6-44

Interrupt vector assignments.

Remember, locations  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$  are in the read-only-memory. Therefore, the designer must provide the proper reset vector at the time he is writing the monitor program.

Figure 6-45 shows the sequence of events that occurs when the MPU is reset. First, the interrupt (I) mask bit is set. You will recall that the (I) flag is one of the condition code registers. As you will see later, if this flag is set, it prevents one of the other interrupts from occurring. Thus, the MPU sets the interrupt mask bit so that the reset sequence will not be interrupted by a request for interrupt by one of the I/O devices.

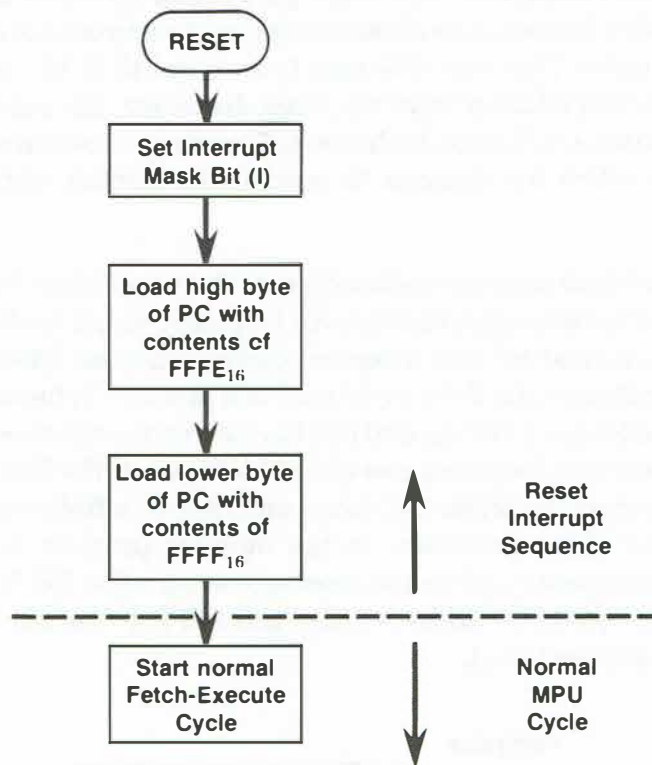


Figure 6-45

Reset interrupt sequence.

Second, the contents of location  $\text{FFFE}_{16}$  are loaded into the high byte of the program counter. This is done by sending the address  $\text{FFFE}_{16}$  out on the address bus. The memory location is read out, and its contents are placed on the data bus. The MPU picks up this byte and places in the upper eight bits of the program counter. In our example, the byte in location  $\text{FFFE}_{16}$  is  $\text{FC}_{16}$ .

Next, the contents of location  $FFFF_{16}$  are loaded into the lower eight bits of the program counter. This is done by setting the address bus to  $FFFF_{16}$ . Thus, the contents of the highest memory location are placed on the data bus. In our example, this byte is  $00_{16}$ . At this point, the program counter contains the address of the first instruction which is  $FC00_{16}$ .

The reset sequence is then terminated by switching the MPU to its normal fetch-execute machine cycle. Thus, the instruction at address  $FC00_{16}$  is fetched and executed. From this point on, all MPU activities are controlled by the program.

The microprocessor system will have a reset switch somewhere in the system. This will allow the operator to restart the system if the system locks up or runs away for some reason. In addition, some systems will have an automatic feature that will allow the system to reset itself after a power failure. In both cases, the reset capability of the MPU is used.

The reset capability can be considered an interrupt, since the MPU leaves whatever it is doing and jumps off to the start of the monitor program. In most cases, the monitor program would start with a short subroutine that initializes the system. It would do things like set up the stack pointer, initialize displays, etc.

## Non-Maskable Interrupts

The 6808 has two other types of hardware interrupts. One of these interrupts is maskable; the other is not. A maskable interrupt is one that the MPU can ignore under certain conditions. Whereas, a non-maskable interrupt cannot be ignored. To illustrate the difference, recall the corporation president analogy.

The president's report writing can be interrupted by the telephone; however, by telling her secretary to hold all calls, she has effectively masked one source of interruptions. In this analogy, it is impractical to mask all interrupts. For example, it would be counterproductive to mask the fire alarm and have the building burn down.

The same situation can exist, to some extent, in a microprocessor-controlled system. Some interrupts can be ignored for a few seconds while the MPU is performing a more important task. This type of interrupt can be masked. Other interrupts must not be ignored at all. These interrupts cannot be masked. Of course, it is up to the designer to decide which interrupts can be masked and which cannot. The 6808 MPU has provisions for handling both types. How the MPU handles the non-maskable type will be discussed first.

The 6808 MPU has a control line called the non-maskable interrupt ( $\overline{\text{NMI}}$ ) line. A high-to-low transition on this line forces the MPU to initiate a **non-maskable interrupt sequence**. The purpose of this sequence is to provide an orderly means by which the MPU can jump off to a service routine that will take care of the interrupt.

This becomes somewhat involved because the MPU must be able to go back to its main program after the interrupt service routine is finished. It must be able to pick up exactly where it left off. Furthermore, all registers must hold exactly the same data and addresses that they held when the interrupt occurred. In other words, when an interrupt occurs, the program count must be saved so that the MPU can later return to this point in the program. Also, the contents of the accumulators, index register, and the condition code registers must be saved so that the MPU can be restored to the exact condition that existed at the instant the interrupt occurred.

The 6808 MPU accomplishes this by pushing all the pertinent data onto the stack. Then, after the interrupt has been serviced, the MPU returns to its previous status by pulling the data back from the stack.

The non-maskable interrupt sequence is shown in Figure 6-46. A non-maskable interrupt is initiated when the  $\overline{\text{NMI}}$  line goes from its high state to its low state. The MPU finishes the execution of the current instruction; however, before another instruction is fetched, the MPU pushes the contents of its registers onto the stack. Recall that the stack pointer always points to the top of the stack. For this example, assume that the stack pointer was set by an earlier instruction to address  $0068_{16}$ .

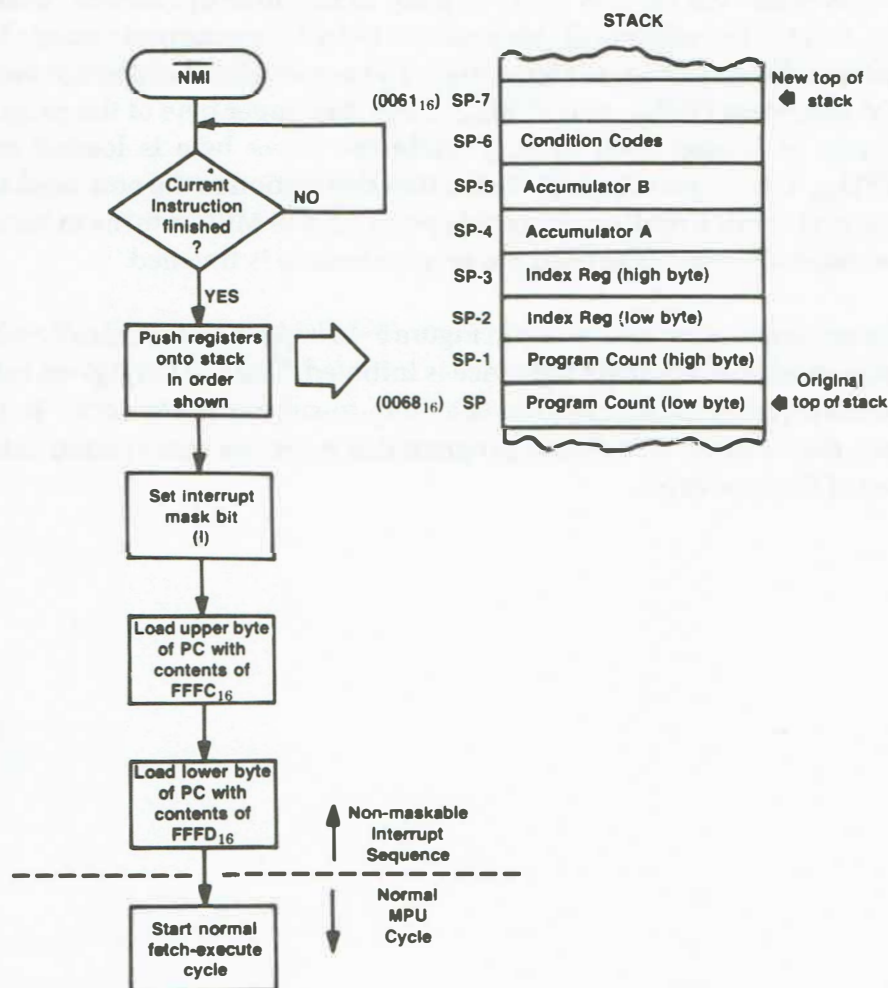


Figure 6-46

Non-maskable interrupt sequence.



The MPU pushes the lower eight bits of the program counter into memory location  $0068_{16}$ . Then, it decrements the stack pointer so that the upper eight bits of the program counter are pushed into address  $0067_{16}$ . Next, the contents of the index register are pushed into address  $0066_{16}$  and  $0065_{16}$ . The contents of accumulators A and B and the condition codes are also pushed as shown. When all this has been done, the stack pointer will have been decremented seven times to  $0061_{16}$ .

Return to the flow chart of Figure 6-46 and notice that the next step is to set the interrupt mask bit. This allows the MPU to ignore any interrupt requests that occur while the non-maskable interrupt is being serviced.

At this point, the MPU is ready to jump to the interrupt service routine. But, what is the address of this routine? Recall the interrupt vector chart that was shown earlier, in Figure 6-44. The non-maskable interrupt vector is at addresses  $FFFC_{16}$  and  $FFFD_{16}$ . Thus, the upper byte of the program counter is loaded from  $FFFC_{16}$  while the lower byte is loaded from  $FFFD_{16}$ . This directs the MPU to the first instruction in the non-maskable interrupt service routine. From this point on, the MPU returns to its normal fetch-execute cycle until the service routine is finished.

The sequence of events shown in Figure 6-46 happen automatically when a non-maskable interrupt sequence is initiated. The  $\overline{\text{NMI}}$  line gives external hardware a method of forcing a jump-to-subroutine to occur. In this case, the subroutine is a short program that performs some action to take care of the interrupt.

## Return-From-Interrupt (RTI) Instruction

The non-maskable interrupt is used when some situation exists that cannot be ignored. You can probably visualize applications that would require such a capability. For example, assume that a microprocessor is being used in a numerically-controlled drill press. The non-maskable interrupt could be used in conjunction with limit switches to prevent drilling holes in the work surface. Or, it could be used to shut down the machine if someone's hand got too close.

The purpose of the service routine is to direct the operation of the computer to take care of the interrupt. Typically, it would first determine which external device initiated the interrupt. Then, it would determine the nature of the interrupt. Finally, it would take whatever action was necessary to take care of the interrupt. In many cases, the interrupt is of a routine nature and can be easily serviced. In these situations, the MPU should return to the main program and take up where it left off. There is an instruction that allows the MPU to do this. It is called the "Return-from-Interrupt" (RTI) instruction. Look at your Instruction Set Summary card, and you will see that this is a 1-byte instruction whose opcode is  $3B_{16}$ .

Figure 6-47 shows how the RTI instruction is used. The main program is shown on the left, while the interrupt service routine is shown on the right. Assume that the interrupt signal occurs while the LDAB# instruction is being executed. The MPU finishes that one instruction and pushes all pertinent data onto the stack. It then jumps to an address determined by the  $\overline{NMI}$  vector in address  $FFFC_{16}$  and  $FFFD_{16}$ . The contents of these two locations determine the starting address of the  $\overline{NMI}$  service routine. Notice that the last instruction in the service routine is the Return-from-Interrupt instruction. This instruction returns program control to the point in the main program that the MPU left when the interrupt occurred.

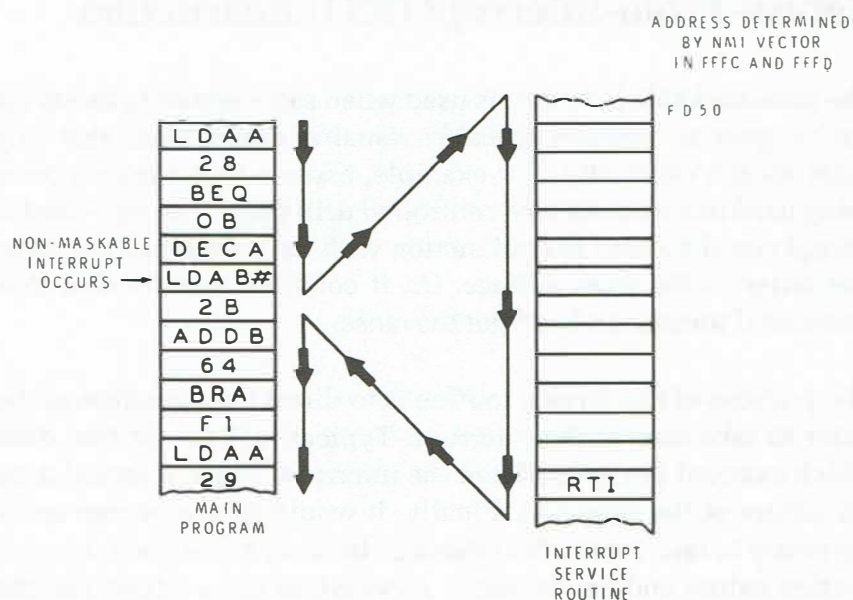


Figure 6-47

The RTI instruction returns control to the main program after the interrupt has been serviced.

This can be done because the previous status of the MPU was preserved in the stack. The RTI instruction causes the accumulators, index registers, condition code register, and the program counter to be loaded from the stack. Thus, the same information that went into the stack when the interrupt occurred comes out of the stack when the RTI instruction is executed. This allows the MPU to return to the main program and take up where it left off.

## Interrupt Request (IRQ)

The interrupt request is very similar to the non-maskable interrupt. The main difference between the two is that the interrupt request is maskable.

The 6808 MPU has a control line called the interrupt request ( $\overline{\text{IRQ}}$ ) line. When this line is low, an interrupt sequence is requested. However, the MPU may or may not initiate the sequence, depending on the state of the interrupt mask (I) bit in the condition code register. If the (I) bit is set, the MPU ignores the interrupt request. If the (I) bit is not set, the MPU initiates the interrupt sequence. The procedure is very similar to the  $\overline{\text{NMI}}$  procedure discussed earlier. Figure 6-48 shows the  $\overline{\text{IRQ}}$  interrupt procedure.

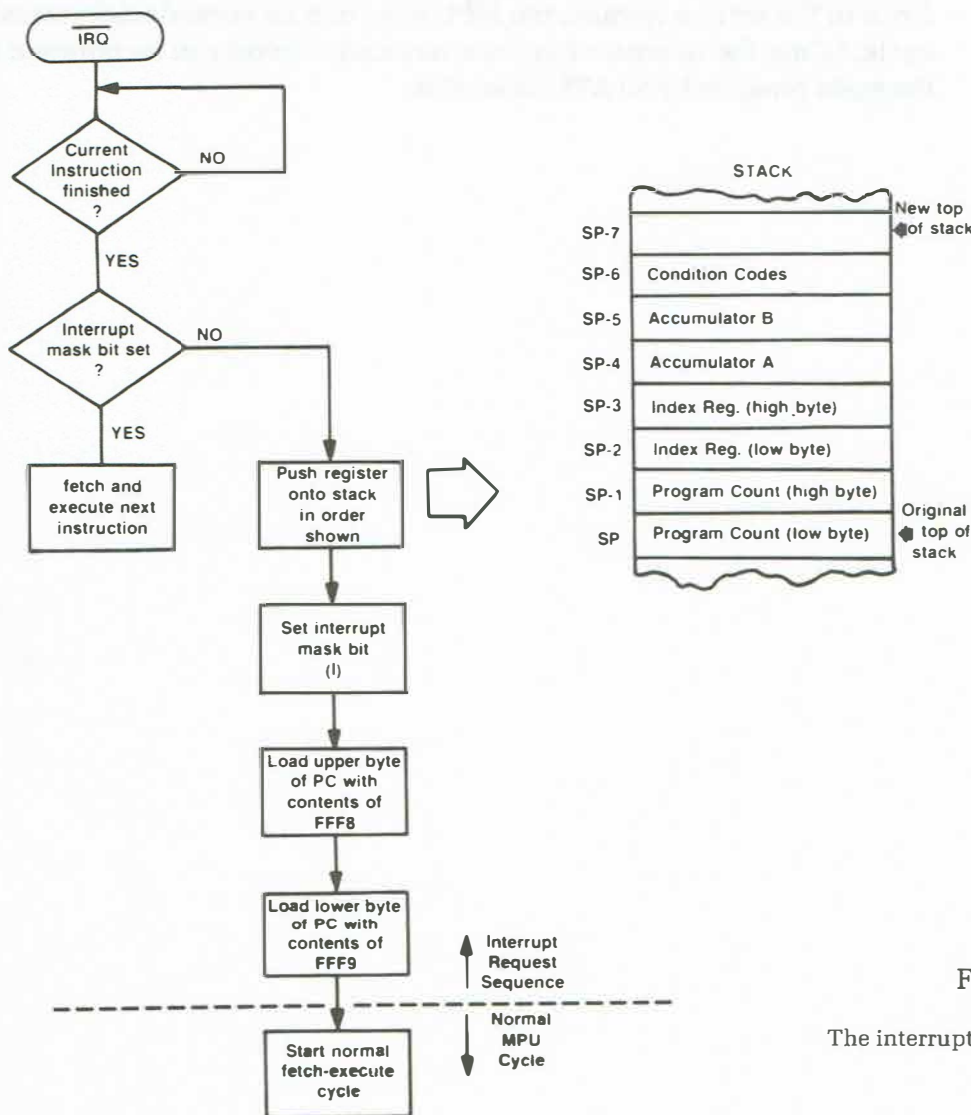


Figure 6-48

The interrupt request ( $\overline{\text{IRQ}}$ ) sequence.

When the  $\overline{\text{IRQ}}$  line is low, the MPU finishes the current instruction. It then checks the interrupt mask bit. If (I) is set to 1, the MPU ignores the interrupt request and executes the next instruction in sequence. However, if (I) = 0, the MPU pushes the contents of the various registers onto the stack in the order shown.

Next the interrupt mask bit is set to 1. This prevents the MPU from honoring other interrupt requests until the present interrupt has been serviced.

The address of the  $\overline{\text{IRQ}}$  service routine is at addresses  $\text{FFF8}_{16}$  and  $\text{FFF9}_{16}$ . The program counter is loaded from these addresses. Thus, the next instruction to be executed will be the first instruction in the interrupt request service routine.

Once in the service routine, the MPU goes into its normal fetch-execute cycle. When the interrupt has been serviced, control can be returned to the main program by an RTI instruction.



## Interrupt Mask Instructions

The 6808 MPU has two instructions that allow software control of the interrupt mask bit. You have seen that the (I) bit in the condition code register is set any time an interrupt sequence is initiated. This prevents an  $\overline{\text{IRQ}}$  from being honored while a previous  $\overline{\text{IRQ}}$  or  $\overline{\text{NMI}}$  is being serviced. This is an example of setting the interrupt flag with hardware.

In many cases, it is necessary to set the interrupt flag with software. Therefore, the 6808 MPU has an instruction that can do this. It is called the “Set-Interrupt-Mask” (SEI) instruction. If you refer to your Instruction Set Summary card, you will see that this is a 1-byte instruction whose opcode is  $0F_{16}$ . The flag may be set to prevent an interruption on a part of the program that we do not wish to be interrupted. It has the effect of disabling interrupt requests.

Of course, we do not wish to permanently disable the interrupt capability; therefore, some means must be provided for enabling the interrupt request capability. An instruction called “Clear-Interrupt-Mask” (CLI) is available for this purpose. This is a 1-byte instruction whose opcode is  $0E_{16}$ .

While we can disable or enable the interrupt request line with these instructions, they do not affect the non-maskable interrupt. As the name implies, the NMI line cannot be disabled by the (I) flag.

## Software Interrupt (SWI) Instruction

The 6808 MPU has a software equivalent of an interrupt. It is an instruction called the "Software Interrupt" (SWI). When executed, the instruction causes the MPU to perform an interrupt sequence that is very similar to the hardware interrupt sequences already discussed. As shown on your Instruction Set Summary card, this is a 1-byte instruction whose opcode is  $3F_{16}$ .

Figure 6-49 shows the sequence of events that occurs when this instruction is executed. First, the contents of all the pertinent registers are pushed onto the stack in the order shown. Next, the interrupt mask is set so that interrupt requests cannot interfere. Finally, the software interrupt vector is obtained from addresses  $FFFA_{16}$  and  $FFFB_{16}$ . This vector is loaded into the program counter so that the next instruction will be fetched from this address. As with other interrupts, the MPU will return to the original program when a Return-from-Interrupt instruction is encountered.

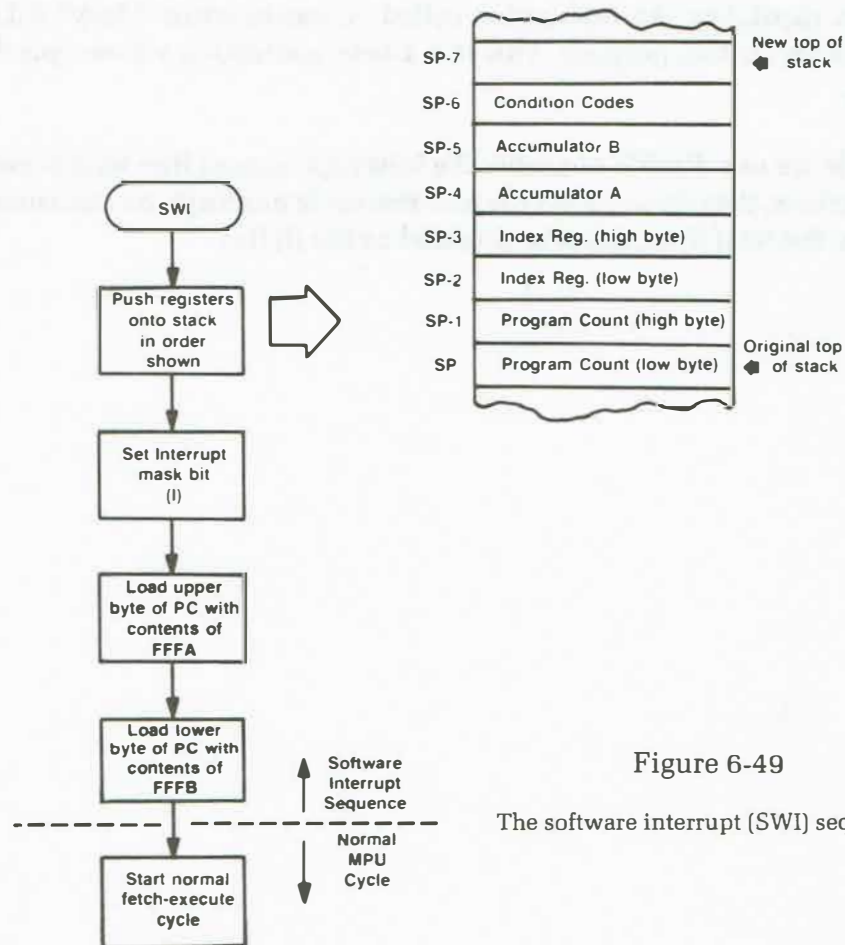


Figure 6-49

The software interrupt (SWI) sequence.

The software interrupt instruction can be used to simulate hardware interrupts. It is also helpful for inserting pauses in a program.

## Wait For Interrupt (WAI) Instruction

One of the first instructions you were introduced to was the halt instruction, opcode  $3E_{16}$ . Previously, you learned that this instruction is actually called a “Wait-for-Interrupt” (WAI). What exactly does this instruction do? It does cause the MPU to halt, but there is more to it than that.

When the WAI instruction is executed, the program counter is incremented by one. Then the contents of the program counter, index register, accumulators, and condition code register are pushed onto the stack. The order is exactly the same as if an interrupt occurs. The MPU then enters a wait state, doing nothing further unless an interrupt occurs.

The MPU can be forced back into action either by an interrupt request or by a non-maskable interrupt. The  $\overline{NMI}$  sequence is the same as that described earlier except for one important difference. Remember, the contents of the registers have already been pushed onto the stack. Thus, this part of the  $\overline{NMI}$  sequence is omitted. This allows the MPU to respond faster to the interrupt.

The  $\overline{IRQ}$  sequence is also the same as that described earlier except that the registers are not pushed onto the stack again. As always, the  $\overline{IRQ}$  signal is ignored if the interrupt mask bit is set. Of course, the reset signal can override the wait state; thus, there are three ways of escaping the wait state.

## Programmed Review

57.	The 6808 MPU has _____ different types of interrupts.
58.	(four) The Interrupt Request ( <u>IRQ</u> ) _____ ignored by the MPU if the interrupt mask bit is set. (is/is not)
59.	(is) The main purpose of the _____ interrupt sequence is to load the address of the first instruction of the main program to be executed into the program counter.
60.	(reset) An interrupt _____ is an address that is loaded into the program counter when an interrupt occurs.
61.	(vector) The reset sequence is terminated by switching the MPU to its normal _____ - _____ machine cycle.
62.	(fetch-execute) A _____ interrupt cannot be ignored by the MPU. (maskable/non-maskable)
63.	(non-maskable) Once a routine interrupt has been serviced, execution of the _____ instruction allows the MPU to return to the main program and take up where it left off.
64.	(RTI) The Set-Interrupt-Mask (SEI) instruction allows for _____ control of the interrupt mask bit. (hardware/software)

65. (software) A software interrupt instruction \_\_\_\_\_  
(can/cannot)  
be used to simulate a hardware interrupt.

66. (can) When the Wait-for-Interrupt (WAI) instruction is executed,  
the program counter is \_\_\_\_\_ by one.  
(incremented/decremented)

67. (incremented) If the MPU is waiting for an interrupt, the  $\overline{\text{IRQ}}$  and  
 $\overline{\text{NMI}}$  interrupts \_\_\_\_\_ cause data to be pushed into  
the stack.  
(will/will not)

(will not)



## EXPERIMENTS

Perform Programming Experiments 11 and 12. You will find these experiments in Unit 12. After you finish these experiments, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your knowledge of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct answers that appear after the exam.

1. Which of the following program segments will NOT clear both accumulators?
  - A. CLRA  
CLRB
  - B. CLRA  
TAB
  - C. CLRB  
TBA
  - D. CLRA  
ABA
2. Which of the following contains an operation that can NOT be performed directly on a byte in memory using a single instruction?
  - A. Increment, decrement, shift left arithmetically.
  - B. Clear, complement, compare.
  - C. Rotate left, negate, test for zero.
  - D. Shift right logically, rotate right, test for minus.
3. Which addressing mode is best suited for adding a list of numbers?
  - A. Direct.
  - B. Extended.
  - C. Indexed.
  - D. Relative.
4. Which of the following program segments will successfully swap the contents of accumulators A and B?

A. TAB TBA	C. TAB ABA
B. STAA 10 TBA LDAB 10	D. STAA 10 LDAB 10 TBA

5. Which of the following instructions can be used to clear the (Z) flag?
- A. BEQ.
  - B. BNE.
  - C. NOP.
  - D. TAP.
6. Which of the following instructions can be used to test the result of the subtraction of unsigned binary numbers?
- A. BGE.
  - B. BGT.
  - C. BCS.
  - D. BLT.

NOTE: Refer to the program shown in Figure 6-50 for questions 7 through 13.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	4F	CLRA	Clear Accumulator A.
0011	7D	TST	Test
0012	00	00	the
0013	1E	1E	multiplier.
0014	27	BEQ	If it is zero branch to wait.
0015	07	07	
0016	7A	DEC	Otherwise decrement
0017	00	00	the
0018	1E	1E	multiplier.
0019	9B	ADDA	Add the
001A	1F	1F	multiplier to the product.
001B	20	BRA	Repeat the loop.
001C	F4	F4	
001D	3E	WAI	Wait.
001E	05	Multiplier	
001F	04	Multiplicand	

Figure 6-50

This program multiplies by repeated addition.

7. What addressing mode does the TST instruction use?
  - A. Immediate.
  - B. Direct.
  - C. Extended.
  - D. Indexed.
8. The BEQ instruction checks to see if the TST instruction has set the:
  - A. (Z) flag.
  - B. (C) flag.
  - C. (H) flag.
  - D. (V) flag.
9. The DEC instruction decrements the number in:
  - A. Accumulator A.
  - B. Memory location 001E.
  - C. Accumulator B.
  - D. The index register.
10. Which instruction is executed immediately after the BRA instruction?
  - A. WAI.
  - B. BEQ.
  - C. CLRA.
  - D. TST.
11. With the values given for the multiplier and multiplicand, how many times will the main program loop be repeated?
  - A. Four times.
  - B. Five times.
  - C. Twenty times.
  - D. Two times.
12. After the program has been executed, memory location 001E will contain:
  - A.  $05_{16}$ .
  - B.  $04_{16}$ .
  - C.  $20_{16}$ .
  - D.  $00_{16}$ .
13. After the program has been executed, the product will appear in:
  - A. Memory location 001E.
  - B. Memory location 001F.
  - C. Accumulator A.
  - D. Accumulator B.

14. If the (I) bit in the condition code register is set, the MPU will ignore:
  - A. The reset signal.
  - B. The non-maskable interrupt signal.
  - C. The interrupt request signal.
  - D. The software interrupt instruction.
15. Which of the following lists contains instructions that DO NOT change the contents of the stack pointer?
  - A. PULA, DES, RTI, WAI.
  - B. PSHB, INS, RTS, SWI.
  - C. TXS, BSR, PULB, LDS.
  - D. PSHA, JMP, TSX, STS.
16. The 6808 MPU stack pointer is automatically:
  - A. Decrement before data is pushed onto the stack.
  - B. Increment before data is pushed onto the stack.
  - C. Decrement after data is pushed onto the stack.
  - D. Increment after data is pushed onto the stack.
17. One difference between the JMP and JSR instruction is:
  - A. JMP can use either extended or indexed addressing.
  - B. The program count is saved when JSR is executed.
  - C. The JSR will be executed even if the interrupt mask is set.
  - D. JMP is an unconditional jump.
18. The last instruction in a subroutine is generally:
  - A. A JMP instruction.
  - B. An RTS instruction.
  - C. An RTI instruction.
  - D. A JSR instruction.
19. Which of the following types of interrupts DOES NOT cause data to be pushed into the stack?
  - A. Software interrupt.
  - B. Non-maskable interrupt.
  - C. Reset interrupt.
  - D. Interrupt request.



20. Generally, the last instruction in an interrupt service routine will be:
- A. An RTI instruction.
  - B. An SWI instruction.
  - C. An RTS instruction.
  - D. An NMI instruction.

It is better to have a hundred small businesses than a few large ones.

Small businesses are the backbone of the economy.  
They create jobs and provide services.  
They are the lifeblood of the community.  
They are the heart of the nation.

## EXAMINATION ANSWERS

1. D — The result is that both accumulators will contain whatever was in accumulator B when this program was executed.
2. B — The compare operation cannot be performed directly on a byte in memory.
3. C — Generally, indexed addressing is best suited for adding a list of numbers.
4. B — This program segment swaps the contents of accumulators A and B by using address 10 as a temporary storage location.
5. D — The TAP instructions can be used to set the condition of all flags at once.
6. C — The carry-borrow flag is set when a borrow occurs. Thus, the BCS instruction can be used to determine if the unsigned subtrahend was larger than the unsigned minuend.
7. C — Extended.
8. A — The BEQ instruction tests the Z flag.
9. B — The DEC instruction decrements the number in memory location 001E.
10. D — The relative address (F4) directs the program back to the TST instruction.

11. B — The multiplier (05) is decremented on each pass until it reaches 00. Thus, the loop will be repeated five times.
12. D — The multiplier is reduced to 00 as the program is executed.
13. C — The product appears in accumulator A.
14. C — The MPU will ignore an interrupt request if the I bit of the condition code register is set.
15. D — Neither the JMP, TSX, or STS change the contents of the stack pointer.
16. C — The stack pointer is decremented after data is pushed into the stack.
17. B — The program count is saved when JSR is executed. It is not saved when JMP is executed.
18. B — The last instruction in a subroutine is generally an RTS instruction.
19. C — The reset interrupt does not cause data to be pushed onto the stack.
20. A — Generally, the last instruction in an interrupt sequence will be an RTI instruction.

*Unit 7*

**DATA ACQUISITION**



## CONTENTS

Introduction .....	7-3
Unit Objectives .....	7-4
Unit Activity Guide .....	7-5
Optoelectronic Sensors .....	7-6
Temperature Sensing .....	7-24
Ultrasonic Sensors .....	7-38
Experiment .....	7-48
Unit Examination .....	7-49
Examination Answers .....	7-55

## INTRODUCTION

Until now, you have studied basic robot nomenclature and construction; the various methods of powering and positioning a robot in space; and the controlling MPU, or brain section of the robot. Even with these capabilities, the robot still remains deaf, dumb, and blind; therefore, it is unable to function freely in the outside world. In order to achieve freedom of movement and perform useful functions, the robot must be able to sense and act upon its surrounding environment.

The truly free mobile robot must be able to navigate, avoiding objects in its path. To navigate freely, the robot must be aware of how it arrived at a given point by remembering such data as direction, speed, and distance of travel. Once the robot has reached a given point, it will have to perform a specific task. Again, depending upon the task, the robot will require certain information in order to perform the task. This information could include such parameters as motion, temperature, pressure, proximity, etc.

In this unit, you will study some of the methods that are used to sense the various parameters a robot must monitor in order to perform a certain function. In addition, you will learn how a robot is able to navigate on its own, and, once it has reached a given point, remember how it arrived there.

Examine the Unit Objectives listed in the next section to see what you will learn in this unit. Then follow the instructions in the Unit Activity Guide to be sure you perform all of the steps necessary to complete this unit successfully. Check off each step as you complete it and, in the spaces provided, keep track of the time you spend on each activity.

## UNIT OBJECTIVES

When you complete this unit, you will be able to:

1. Describe how the semiconductor material used in the construction of an LED determines the wavelength of the radiated light.
2. Explain the operation of a phototransistor detector.
3. State the differences between a phototransistor and a photo-Darlington detector, in terms of sensitivity, output, and response time.
4. Explain the operation of an interrupter module.
5. Describe how the focal point of a reflector module can be used to sense slight vibrations.
6. Convert temperature from one scale to another.
7. Determine which type of temperature sensor would provide the greatest accuracy.
8. Explain the difference between negative and positive temperature coefficients.
9. Describe the basic theory of operation of a resistive temperature detector.
10. Explain how a material's elasticity will affect the velocity of ultrasonic waves traveling through the material.
11. Define the term interface in regards to sound propagation.
12. Describe the operation of a piezoelectric transducer.
13. Explain how temperature affects the velocity of ultrasonic waves traveling through air.
14. Describe how ultrasonic resonance can be used to measure the thickness of a material.

# UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "Optoelectronic Sensors."	_____
<input type="checkbox"/> Answer Programmed Review Questions 1-13.	_____
<input type="checkbox"/> Read "Temperature Sensing."	_____
<input type="checkbox"/> Answer Programmed-Review Questions 14-28.	_____
<input type="checkbox"/> Read "Ultrasonic Sensors."	_____
<input type="checkbox"/> Answer Programmed-Review Questions 29-39	_____
<input type="checkbox"/> Perform Experiment 13.	_____
<input type="checkbox"/> Complete The Unit Examination.	_____
<input type="checkbox"/> Check The Examination Answers.	_____

## OPTOELECTRONIC SENSORS

Optoelectronic sensing is one of the most widely used sensory systems found in industry, and the field of robotics is no exception. This is primarily because light sensing systems are inexpensive, reliable, and accurate. In addition, optoelectronic sensory systems can be used to monitor: the speed, direction, and amount of angular and linear motion; the presence of an object in a specific area; the thickness of some materials; plus, many other parameters associated with industrial processing and robot control.

An optoelectronic sensing system consists of two main components: the transmitter, or emitter, which produces the light, and the receiver, or detector, which senses the light. Most optoelectronic sensors use light-emitting diodes (LEDs), since silicon light detectors are most sensitive to infrared light and LED's produce infrared light more efficiently than visible light. Light-detector or receiving elements can be either photodiodes, phototransistors, photo-Darlington's, or selenium cells. However, since selenium cells respond slowly to light, they are generally not used to sense motion.

For our discussion on optoelectronic sensing, we will use an infrared light-emitting diode (IRED) as the emitter and a phototransistor as the detector. We will first look at each of the components separately; then we will see how the two components are combined into a single unit to produce an optoelectronic sensor.

### Light-Emitting Diodes

An LED is a solid-state component, and like all solid-state devices, has a nearly unlimited life expectancy. It is physically more sturdy than an incandescent or neon lamp and requires less operating power. Since it is such an important component of the optoelectronic sensing system, we will examine the operation and construction of an LED in some detail.



## LED OPERATION

An LED is basically a PN junction diode. All semiconductor diodes emit energy in response to an electric current. The emission may be in the form of heat energy, light energy (photons), or both, and is caused by a recombination of electrons and holes. The radiated wavelength is a function of the electron's energy level and is determined by the type of semiconductor material used in the construction of the diode.

The LED uses this principle to emit light through the recombination of electrons and holes when current is forced through its junction as illustrated in Figure 7-1. As shown, the LED must be forward biased so that the negative terminal of the battery will inject electrons into the N-type layer (the cathode). These electrons will move toward the PN junction. Corresponding holes will appear at the P-type or anode end of the diode (actually caused by the movement of electrons) and also appear to move toward the junction. The electrons and holes merge toward the junction where they may combine. If the electron possesses sufficient energy when it fills a hole, it will produce a photon of light energy. Many such combinations result in a substantial amount of light (many photons) being radiated from the device in various directions.

By now, you are probably wondering why the LED emits light and an ordinary diode does not. This is simply because most ordinary diodes are made from silicon, and silicon is an opaque, or impenetrable, material as far as light is concerned. Any photons that are produced in an ordinary diode simply cannot escape. LED's on the other hand are made from semiconductor materials that are semitransparent to light energy. Therefore, in an LED, some of the light energy produced can escape.

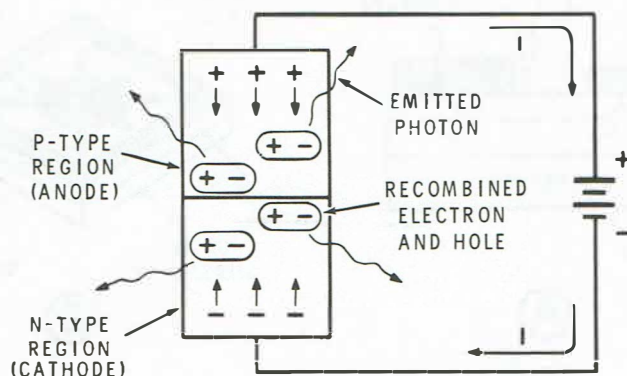


Figure 7-1  
Basic Operation Of A Light Emitting Diode

## LED CONSTRUCTION

Many LED's are made of gallium arsenide (GaAs). LED's constructed with gallium arsenide emit light most efficiently at a wavelength of approximately 900 nanometers, which is in the infrared region of the light spectrum, and is not visible to the human eye. Other materials are also used, such as gallium-arsenide phosphide (GaAsP) which emits a visible red light at approximately 660 nanometers, and gallium phosphide (GaP), which produces a visible green light at approximately 560 nanometers. Also the GaAsP LED can be made to emit light at any wavelength from approximately 550 nanometers to 910 nanometers by adjusting the percentage of phosphide.

The construction of a typical (GaAsP LED) is shown in Figure 7-2. Figure 7-2A shows a cross-section of the device and Figure 7-2B shows the entire LED chip. The construction begins with a gallium arsenide (GaAs) substrate. Then a layer of gallium arsenide phosphide (GaAsP) crystal is grown on this substrate. The concentration of gallium phosphide (GaP) in this layer is gradually increased from zero to the desired level. A gradual increase is required so that the crystalline structure of the substrate is not disturbed. During this growth period, an N-type impurity is added to make the layer an N-type material. The grown layer is then coated with a special insulative material, and a window is etched into this insulator. A P-type of impurity is then diffused through the window into the epitaxial layer, and the PN junction is formed. The P-type layer is made very thin so that the photons generated at or near the PN junction will have only a short distance to travel through the P-type layer, and can escape through the window as shown in Figure 7-2A.

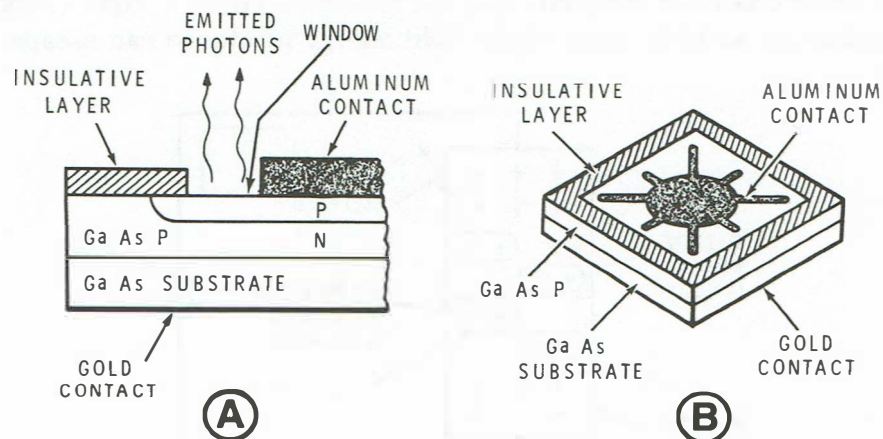


Figure 7-2  
Basic Configuration Of An LED

To complete the construction of the LED, electrical contacts are attached to the P-type region and the bottom of the substrate. The upper contact Figure 7-2B has a number of fingers extending outward so that current will be distributed evenly throughout the junction area.

Next, the LED must be mounted in a suitable package. Several types of packages are commonly used but all must fulfill one important requirement — to optimize the emission of light from the LED. This factor is very important because the LED emits only a small amount of light. Some packages contain a lens system that gathers and, in effect, concentrates the light produced by the LED. Also, various package shapes are used to obtain variations in the width of the emitted light beam.

A typical LED package and schematic symbol are shown in Figure 7-3. As shown, the package body and lens are one piece and are molded from plastic (epoxy). The cathode and anode leads are inserted through the case and extended up into the dome shaped top which serves as the lens. The bottom contact of the LED chip is attached directly to the cathode lead, and the upper contact is connected to the anode lead by a thin wire that is bonded in place. The placement of the LED chip in the case is critical, since the case serves as a lens that conducts light away from the LED and also as a magnifier. In some cases, the plastic lens will contain fine particles that help to diffuse the light.

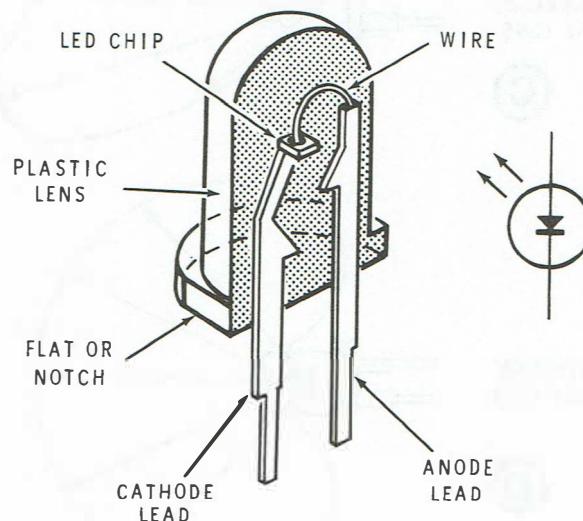


Figure 7-3  
A Typical LED Package and Schematic Symbol

The same LED chip in different types of cases will have much different radiation patterns. The most common differences in lens construction are; lens material (diffused or clear), lens shape, and the distance from the lens surface to the chip. Figure 7-4 shows the effects of these variations upon the radiation pattern. The lens construction shown in Figure 7-4A is most commonly used in optoelectronic sensing systems, where a narrow beam of light is required for greater accuracy.

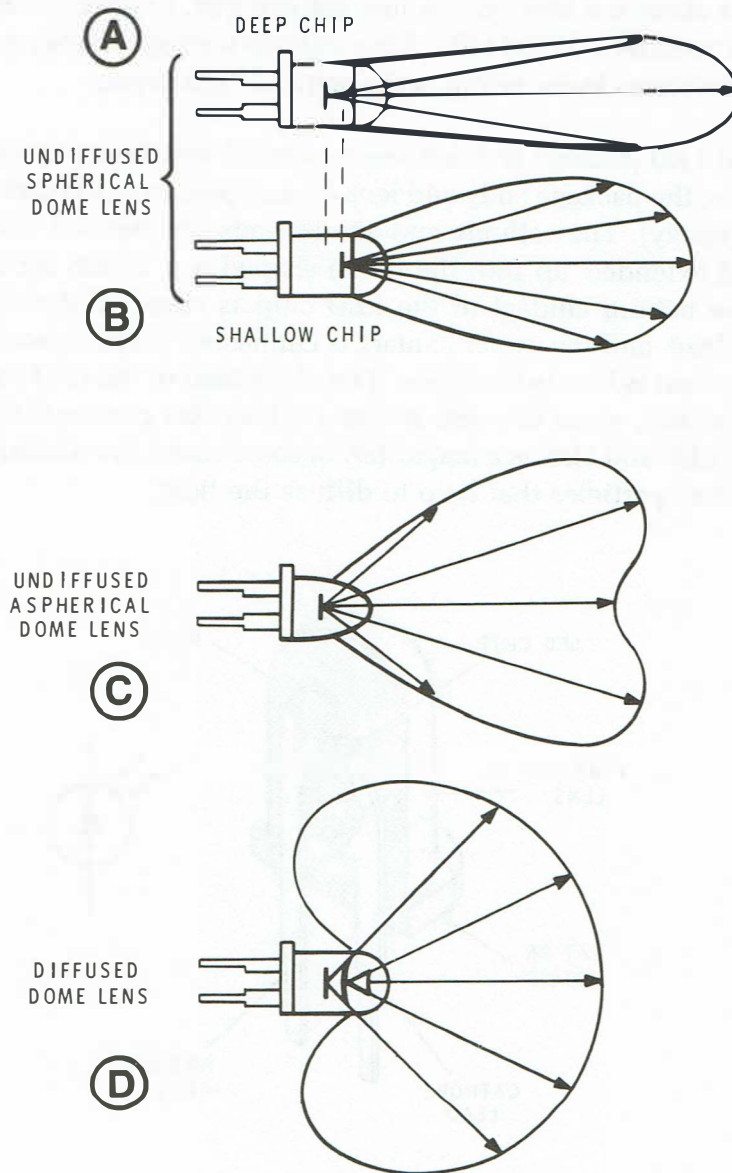


Figure 7-4  
Radiation Pattern Variation of LED Lamps.



## Phototransistors

Whereas, the light-emitting diode was used to emit light; the phototransistor is used to detect light in an optoelectronic sensing system.

### PHOTOTRANSISTOR CONSTRUCTION

The phototransistor is also a solid-state PN junction device. However, it has two junctions instead of one, like the LED just described. The phototransistor, shown in Figure 7-5, is constructed in a manner similar to an ordinary transistor. The process begins by taking an N-type substrate (usually silicon), which ultimately serves as the transistor's collector, and diffusing into this substrate a P-type region which serves as the base. Then an N-type region is diffused into the P-type region to form the emitter. The phototransistor therefore resembles a standard NPN bipolar transistor in appearance. The phototransistor is often packaged much like the standard transistor, with three leads provided which connect to the emitter, base, and collector regions. Other phototransistors, similar to the one shown in Figure 7-6, omit the base lead, using only emitter and collector leads. The phototransistor is physically mounted under a transparent window so that light can strike the surface of its base region.

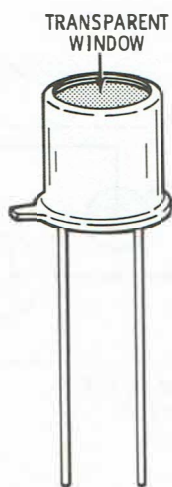


Figure 7-6  
Two-Lead Phototransistor Package

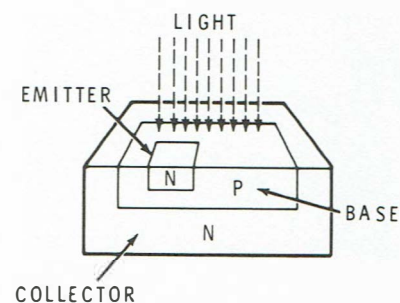


Figure 7-5  
The Construction Of A  
Typical Phototransistor



## PHOTOTRANSISTOR OPERATION

The operation of a phototransistor is easier to understand if it is represented by the equivalent circuit shown in Figure 7-7. Notice that the circuit shows a light-sensitive diode connected across the base and collector junction of a conventional NPN bipolar transistor. If the equivalent circuit is biased by an external source as shown, current will flow into the emitter lead of the circuit and out of the collector lead. The amount of current flowing through the circuit is controlled by the transistor in the equivalent circuit. The transistor conducts more or less depending upon the conduction of the light-sensitive diode, which in turn, conducts more or less as the light striking it increases or decreases in intensity. If light intensity increases, the diode conducts more photocurrent (its resistance decreases) thus allowing more emitter-to-base current (commonly referred to as base current) to flow through the transistor. This increase in base current is relatively small but, due to the transistor's amplifying ability, the small base current is used to control the much larger emitter-to-collector current flowing through this device. Any increase in input light intensity causes a substantial increase in collector current. A decrease in light intensity would cause a corresponding decrease in collector current.

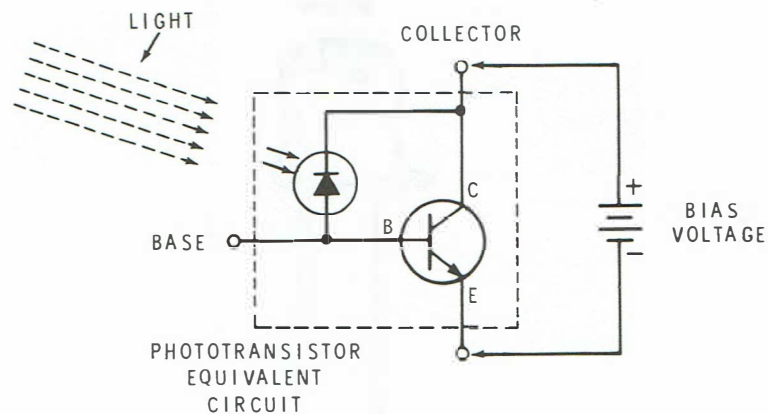


Figure 7-7  
Equivalent Circuit For a Phototransistor

Although the phototransistor may have a base lead as well as emitter and collector leads, the base lead is used in very few applications. When the base lead is used, it is simply subjected to a bias voltage which will set the transistor's collector current to a specific value under a given set of conditions. In other words, the base may be used to adjust the phototransistor's operating point. In most applications, only the emitter and collector leads are used, and the device is considered to have only two terminals.

The phototransistor provides an output current that is essentially controlled by the intensity of the light striking its surface and, to a much lesser degree, by its operating voltage. The phototransistor's sensitivity to light is determined by setting its collector voltage to a specific value and allowing the device to vary its collector current in accordance with the changes in light intensity.

The phototransistor can produce a much higher output current than other photosensitive devices for a given light intensity because the phototransistor has a built-in amplifying ability. The phototransistor's higher sensitivity makes it useful for a wide range of applications. Unfortunately, this higher sensitivity is offset by one important disadvantage. The phototransistor does not respond rapidly to changes in light intensity, and therefore is not suitable for applications where an extremely fast response is required. Like other photosensitive devices, the phototransistor is used in conjunction with a light source to perform many useful functions. It can be used in place of photoconductive cells and photodiodes in many applications with an improvement in operation. Phototransistors are widely used in such sensing applications as tachometers, smoke and flame detectors, counters, and, as you will see shortly, the eyes of a robot.

A phototransistor is often represented by one of the symbols shown in Figure 7-8A, and it is usually biased as shown in Figure 7-8B. As shown, the phototransistor is used to control the current flowing through a load.

The phototransistor may also be used as a photodiode by using only its collector and base leads and leaving the emitter open. When it is used in this manner, the PN junction between the collector and base region serves as a photodiode. However, when it is used as a photodiode, the device can produce only a relatively small current, although it will operate at a faster rate.

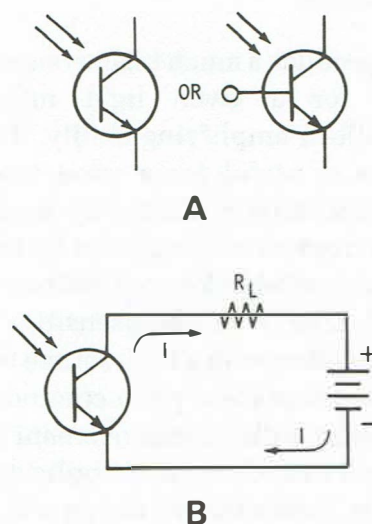


Figure 7-8  
A Phototransistor Schematic Symbol (A)  
Properly Biased Phototransistor (B)

A phototransistor may be interconnected with an ordinary bipolar transistor in a way which allows the phototransistor to control the operation of the bipolar transistor. This type of arrangement is referred to as a photo-Darlington circuit. Such a circuit can also be simultaneously formed and packaged in a single container. The schematic symbol for a photo-Darlington arrangement (in a single package) is shown in Figure 7-9, along with the necessary external circuitry. The phototransistor responds to the input light intensity and conducts accordingly. Its conduction level controls the base current of the bipolar transistor, which in turn controls the current through the load and the battery.

Such an arrangement offers a tremendous increase in sensitivity since the gain of the phototransistor is effectively multiplied by the gain of the bipolar transistor, thus producing a relatively high output current. However, the higher gain is obtained by sacrificing speed. Although it is more sensitive than the photodiode or phototransistor, it has a slower response to changes in light intensity.

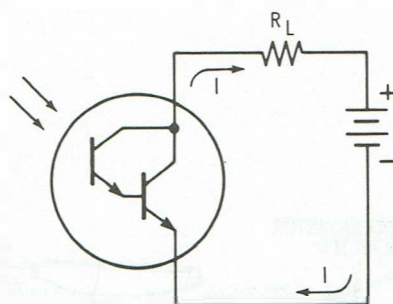


Figure 7-9  
A Properly Biased Photo-Darlington Circuit

## Optoelectronic Sensing System Operation

Figure 7-10 shows a robot-manned parts packaging operation using an infrared emitter and a phototransistor detector configuration as the robot's "eyes". In this case, as long as the infrared light beam is not broken, the phototransistor detector will "sense" the infrared beam and produce a steady, preset, collector current. This steady collector current could be used to provide a data signal to the robot's controller which, in turn, would keep the robot in the stow or ready position.

When a carton moves into the infrared light beam, it will "break" the beam; thus causing no transmitted infrared light to reach the phototransistor detector. The phototransistor will cease conduction; therefore, the data signal will change. The robot's controller will now cause the robot to enter a pre-programmed routine in which it takes a part from the parts conveyor and places it in the carton. By the time the robot has finished its operation, the carton will have passed through the infrared beam. Since the infrared beam is now unbroken, the phototransistor detector will again "sense" the infrared beam. The phototransistor conducts, again providing a data signal to the robot's controller which returns the robot to its stow or ready position, to wait for the next carton.

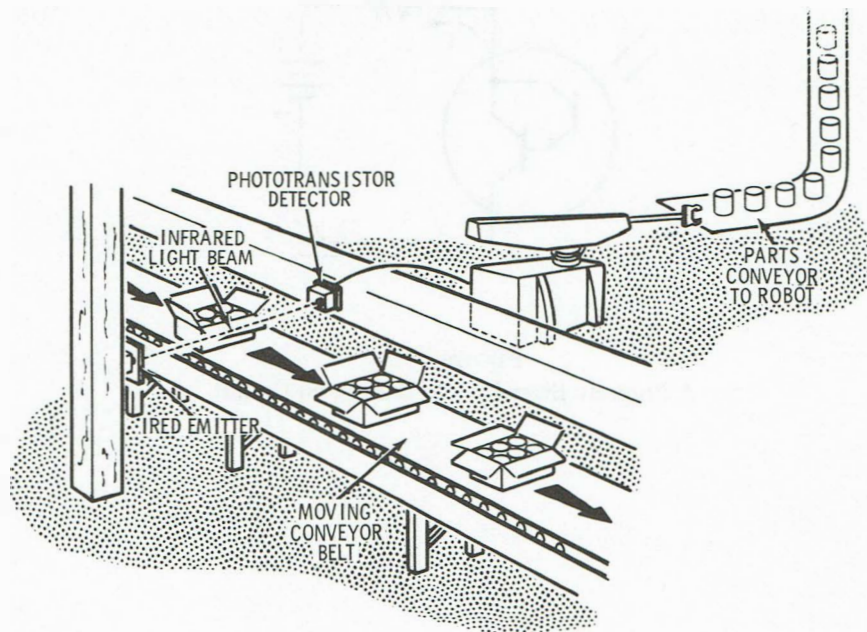


Figure 7-10  
Robot Parts Packing Operation "Using  
Optoelectronic Sensing"



There are several factors that must be considered with this type of sensing operation. The conveyor belt must maintain a constant speed because, once the carton has broken the light beam, it must arrive at a preset point at the same time the robot does. If the robot and the carton do not arrive at the same point at the same time, the robot will still release the part, as determined by the preset program, thus causing the part to miss the carton. The cartons must also be sufficiently spaced so as to give the robot enough time to complete its operation before the next carton arrives. Also, the infrared emitter and the phototransistor detector should be positioned as close together as possible. Dust particles and even the atmosphere have a tendency to attenuate the light beam. The emitter and detector should not be more than about 30 feet apart. In addition, the light beam should be as narrow as possible for greater sensitivity and accuracy.

This is just one example of how an optoelectronic sensing system could be used. It could just as easily be connected to an electronic counter to count the number of objects passing a given point during a specific period of time. It could also be used as a safety device; whereas, if a person came too close to a dangerous robotic operation, he would break the light beam, thus causing the robot to cease operation or sound an alarm.

## **Interrupter/Reflector Modules**

Until now, the optoelectronic emitter and detector have been considered as discrete components in a light-sensing system. This is not usually the case; most sensing operations use optoelectronic interrupter or reflector modules to obtain the desired parameter. Optoelectronic interrupter and reflector modules contain matched emitter/detector pairs molded into a single unit. However, their physical construction and applications are quite different.

## INTERRUPTER MODULE

Figure 7-11 shows both the physical appearance and schematic diagram of a typical interrupter module. Physically, the infrared light-emitting diode (IRED) is mounted on one side of the package and focused, across a narrow slot, upon a photodetector mounted on the other side. This photodetector may be either a phototransistor, as shown, or a photo-Darlington.

In operation, a steady forward current is allowed to flow through the IRED. If the slot is unobstructed, a constant level of radiation will fall upon the photodetector. However, if an object that attenuates or blocks the light beam is placed within the slot, the resultant decrease in photo current can easily be detected in the phototransistor output circuit. The detected signal can be used to start, stop, or change a robot's program. It can also be used to trigger an alarm, supply a count, or perform other functions dictated by the device's application. It can, in some instances, even determine if the thickness of a semitransparent material is within tolerance. For instance, if a semitransparent material were moved between the emitter and the detector of an interrupter module, and if a certain thickness of the material allowed a specific amount of light to pass through it, any deviation in material thickness would cause a corresponding increase or decrease in the amount of light "sensed" by the detector. This increase or decrease in light could be used to trigger a circuit which would automatically stop the operation until the problem was corrected.

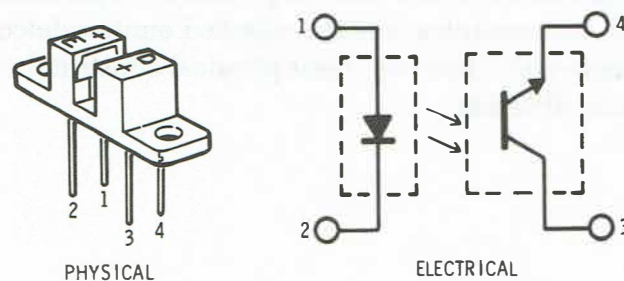


Figure 7-11  
IRED Coupled Interrupter Module

An example of how the interrupter module can be used as an optical tachometer is illustrated in Figure 7-12. The object, of course, is to measure the speed at which the shaft is rotating. A transparent plastic or glass disc is mounted to the shaft, and opaque target areas are painted at evenly-spaced intervals around the perimeter of the disc. An interrupter module is then positioned so that the edge of the disc is centered in the slot. As the opaque targets pass through the infrared light beam, the beam will be blocked periodically, causing pulses to appear in the output circuit. The number of pulses that occur in a given period of time is directly proportional to the speed of rotation.

In this case, since four target areas are used, four output pulses represent one revolution of the shaft. By selecting the proper number of targets, sampling period, and counter scale factor, it is possible to display the rotational speed directly in revolutions-per-minute.

This type of sensing is highly compatible with robotic applications since “sensed” speed or angular rotation, which is normally considered to be an analog function, can be applied directly to the robot’s controller in the form of digital pulses. This alleviates the requirement for an analog-to-digital converter.

Another popular application of the interrupter modules is limit sensing. Mechanical limit switches are subject to malfunction due to friction, contact wear, and mechanical stress. By using interrupters, these problems are avoided since no physical contact is required.

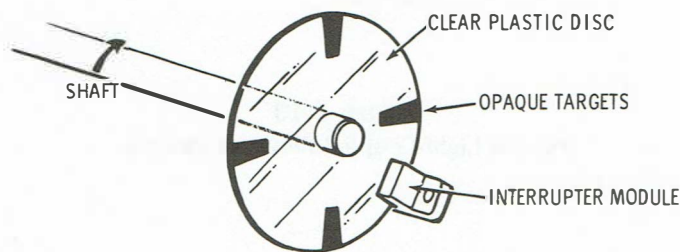


Figure 7-12  
Using an Interrupter Module As An Optical Tachometer

## REFLECTOR MODULES

Reflector modules, as the name implies, operate from reflected radiation rather than from directly transmitted radiation. In this device, an IRED and a phototransistor are mounted side by side and aligned so that their axes converge at a point just beyond the surface of the module. This is shown in Figure 7-13. In order for the phototransistor to receive radiation from the IRED, a reflective surface must be present near the focal point.

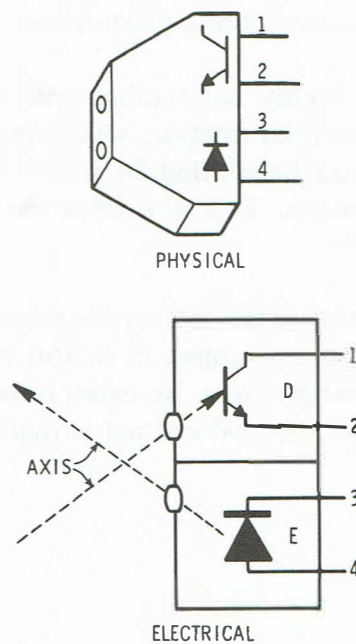


Figure 7-13  
Typical Light-Coupled Reflector Module

To compare the operation of reflector and interrupter devices, we will modify the design of the optical tachometer (shown in Figure 7-12) for use with the reflector module. The modified design is illustrated in Figure 7-14. The surface of the disc has been changed to a dark, light-absorbing finish while the target areas have been made highly reflective. As the disc rotates, very little light will be detected by the phototransistor when a dark area is present at the focal point. However, each time a reflective target reaches this point, a burst of light will be reflected toward the phototransistor, resulting in an output pulse.

Reflector modules are less efficient than interrupter modules, due to their relatively long, high-loss transmission path, but they require access to only one surface of an object. And, reflector modules can be adapted to nearly all of the applications described for interrupter modules. Also, reflector modules can detect slight movements and vibrations, while interrupter modules cannot. When a reflector module is used for this purpose, it is placed so that the object being sensed is precisely at the focal point. Since the light intensity at the detector is a function of the square of the distance to the object, very small changes in distance will result in large changes of output current. Using this method, both the amplitude and frequency can be accurately determined. Also, no physical contact with the object is involved. Therefore, inaccuracies caused by mechanical damping cannot occur, as is possible with mechanical sensors.

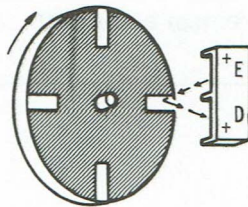


Figure 7-14  
Using a Reflector Module As An Optical Tachometer



## Programmed Review

1.	In an optoelectronic sensing system, the _____ is the source of light.
2.	(transmitter/emitter) Most optoelectronic transmitters transmit _____ light as opposed to visible light.
3.	(infrared) LEDs require _____ operating power than incandescent lamps. (more/less)
4.	(less) LEDs constructed using gallium arsenide emit light most efficiently at a wavelength of approximately _____ nanometers.
5.	(900) Because an LED only emits a small amount of light, a _____ system is often used to gather and magnify the light.
6.	(lens) In an optoelectronic sensing system, a _____ light beam provides the greatest accuracy. (narrow/wide)

7.	(narrow)	A phototransistor is used to _____ light in an optoelectronic sensing system.
8.	(detect/receive)	In a phototransistor, an increase in light will cause _____ in collector current. (an increase/a decrease)
9.	(an increase)	The phototransistor responds relatively _____ to changes in light, when compared to the photo diode. (slowly/quickly)
10.	(slowly)	When an optoelectronic emitter and detector are mounted in the same package and focused across a narrow slot, they are considered to be an _____ module.
11.	(interrupter)	When an opaque object is placed in the slot of an interrupter module, there will be _____ in photo current. (an increase/a decrease)
12.	(a decrease)	Reflector modules operate on the principle of _____ radiation rather than direct radiation.
13.	(reflected)	Reflector devices are _____ efficient than (more/less)
		(less)

## TEMPERATURE SENSING

Temperature sensing is by far the most common type of measurement used in industry today. In fact, of all industrial measured variables, over 50% involve some sort of temperature measurement. Temperature sensing systems can be used for simple “on-off” control, or more elaborate systems can provide extremely accurate, sensitive, and stable control. Temperature sensors are generally classified by their manner of sensing. Some of the more commonly used methods of sensing are bimetallic, resistive, fluid-pressure, thermocouple, radiation, and oscillator.

Before we explore some of the temperature sensing systems in detail, we will first consider the units of temperature measurement. Temperature expressed in degrees Celsius ( $^{\circ}\text{C}$ ) or degrees Fahrenheit ( $^{\circ}\text{F}$ ), is quite common but two other lesser known units of measurement — degrees Kelvin ( $^{\circ}\text{K}$ ) and degrees Rankine ( $^{\circ}\text{R}$ ) — are often used by industry. The absolute temperature scale, shown in Figure 7-15, measures temperature from the theoretically lowest possible temperature value,  $-273.16^{\circ}\text{C}$  or  $-459.69^{\circ}\text{F}$ . When such a scale is based on Celsius or Centigrade divisions, it is called the Kelvin scale. If the absolute scale is based on Fahrenheit division, it is called the Rankine scale. Degrees Kelvin is the official international unit of temperature measurement.

Basic temperature scale conversion is as follows:

$$^{\circ}\text{F} = (^{\circ}\text{C} \times \frac{9}{5}) + 32, \quad ^{\circ}\text{C} = (^{\circ}\text{F} - 32) \times \frac{5}{9}$$

$$^{\circ}\text{R} = ^{\circ}\text{F} + 459.69 \quad ^{\circ}\text{K} = ^{\circ}\text{C} + 273.16$$

	FAHRENHEIT	CELSIUS	KELVIN	RANKINE
ABSOLUTE ZERO	-459.69° F	-273.16° C	0° K	0° R
MELTING POINT OF ICE	32° F	0° C	273.16° K	491.69° R
BOILING POINT OF WATER	212° F	100° C	373.16° K	671.69° R
DIVISION BETWEEN MELTING AND BOILING POINTS OF WATER	180° F	100° C	100° K	180° R

Figure 7-15  
Absolute Temperature Scale (based on a  
pressure of 1 atmosphere)

## Bimetallic Sensors

Bimetallic sensors operate on the principle that different types of metals expand and contract at different rates when they are heated or cooled. The bimetal-strip thermostat, shown in Figure 7-16A, is one of the most common bimetallic sensors. In this case, brass, which has a relatively large coefficient of expansion, and iron, which has a relatively small coefficient of expansion, are fused or bonded together to form the bimetal strip. When the bimetal strip is heated, as shown in Figure 7-16B, the greater expansion of the brass causes the free end of the strip to bend upward. When the heat is removed and the bimetal strip cools, it returns to its normal position. The amount of bimetal-strip deflection is directly proportional to the amount of heat applied.

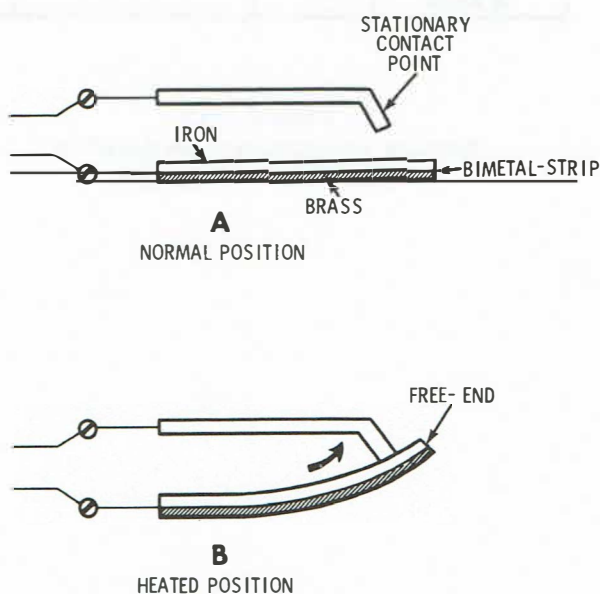
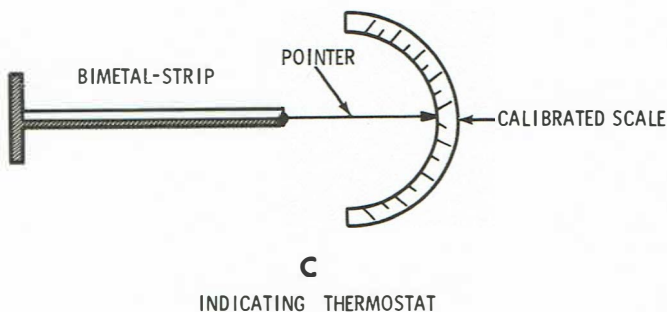


Figure 7-16  
Bimetallic-Strip Thermostats  
(A) Normal Position  
(B) Heated Position  
(C) Indicating Thermostat





An indicating thermostat, shown in Figure 7-16C, has a pointer fastened to its free end, which moves over a calibrated scale, indicating temperature. In addition, a bimetal-strip thermostat can control temperature-related electrical switching, as shown in Figure 7-17A. In this example, an electrical contact is fastened to the free end of a current-carrying bimetal strip so that, when the heat causes the strip to bend a predetermined amount, the movable contact touches the fixed contact, thus completing the control circuit. Again, when the temperature of the bimetal strip decreases, the strip will return to its normal (straight) position, thus separating the electrical contacts and opening the control circuit. To some degree, you can control the amount of heat required to close the circuit, using the adjustment screw on the stationary contact. This adjustment will bring the stationary electrical contact nearer to or farther away from the movable contact. Figure 7-17B is a reverse of this set-up; heat causes the control circuit to open or break.

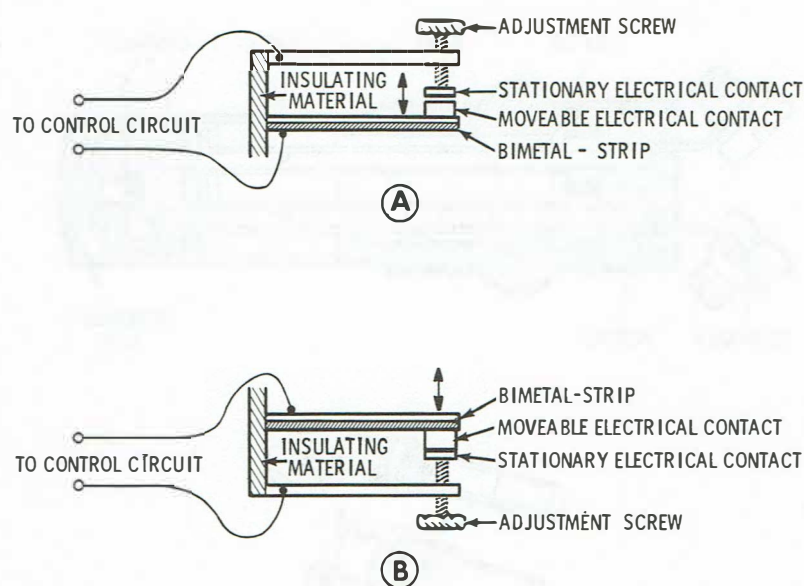


Figure 7-17

Adjustable Bimetal-Strip Switching Thermostats

- (A) Adjustable Bimetal-Strip Thermostat Switch  
(heat will close circuit)
- (B) Adjustable Bimetal-Strip Thermostat Switch  
(heat will open circuit)

Another version of the bimetallic thermostat, called the bimetallic disc switch, is shown in Figure 7-18. Switching is based on a single-pole, single-throw, “snap-action”, bimetallic disc which provides positive and quick make-or-break contacts. When the temperature of the disc changes, the expansion rate of one of the dissimilar metals is much greater than that of the other. When the temperature change is sufficient, the stress created by this unequal expansion overcomes the biasing stress formed in the disc, causing it to invert with a snap. When the temperature changes in the opposite direction, the process is reversed. The temperature sensing element (bimetallic disc) is confined to the very tip to maximize heat sensing through the stainless steel tube.

Whereas the bimetal-strip thermostat can have an adjustable range from  $-40^{\circ}\text{F}$  to  $800^{\circ}\text{F}$ , the bimetallic disc switch has a fixed temperature setting, usually in the range of  $15^{\circ}\text{F}$  to  $525^{\circ}\text{F}$ . The primary advantage of the bimetal-strip and bimetallic disc switch thermostat is their low cost. In addition, the bimetallic disc switch offers a rapid response to either liquid or air temperature.

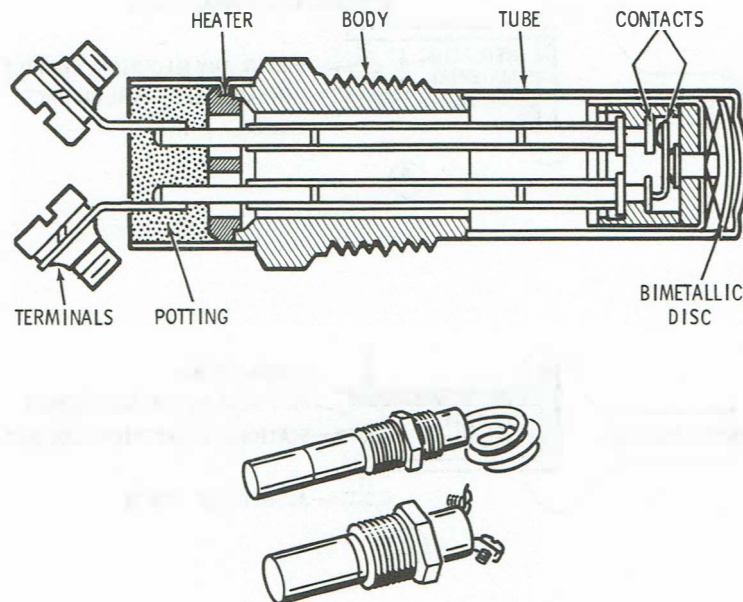


Figure 7-18  
Bimetallic Disc Switch

## Resistive Sensors

There are two basic types of resistive temperature sensors: the conductive type and the semiconductor type; both of which operate on the principle that the resistance of conductors and semiconductors changes with temperature.

### RESISTIVE TEMPERATURE DETECTOR (RTD)

As a rule, the resistance of a metal conductor increases as temperature rises. Therefore, an indication of temperature may be obtained by measuring resistance. The resistive temperature detector (RTD), shown in Figure 7-19, has a sensing element which consists of a coil of fine wire wound on a ceramic core. The fine wire is usually copper, nickel, or platinum; all of which have a positive temperature coefficient — resistance increases as temperature increases.

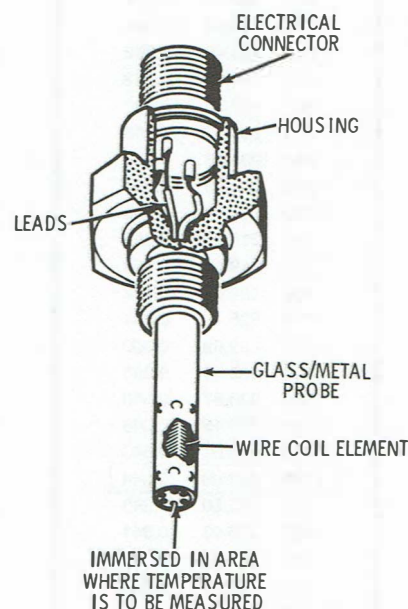


Figure 7-19  
Resistive Temperature Detector (RTD)

Of all usable metals, platinum best meets the requirements of thermometry (temperature measurement). Some of the attributes of platinum are: it can be highly refined (99.999+ % pure); it resists contamination and corrosion; and it is mechanically and electrically stable. In addition, the relationship between temperature and resistance is quite linear; the temperature coefficient of resistance change being approximately .00385% per ohm per °C. See Figure 7-20. Platinum RTDs are used for temperature measurement in the range -270°C to +1100°C; with the maximum temperature for a given RTD being determined by the type of insulation material used to enclose the platinum winding.

Platinum Winding								
°C	Ohm	Ohm/°C	°C	Ohm	Ohm/°C	°C	Ohm	Ohm/°C
220	10.41	0.395	140	153.57	0.375	+500	280.93	0.332
210	14.36	0.417	150	157.32	0.373	510	284.25	0.332
-200	18.53	0.425	160	161.05	0.371	520	287.57	0.330
190	22.78	0.427	170	164.76	0.371	530	290.87	0.329
180	27.05	0.423	180	168.47	0.369	540	294.16	0.327
170	31.28	0.420	190	172.16	0.368	550	297.43	0.327
160	35.48	0.417	+200	175.84	0.367	560	300.70	0.325
150	39.65	0.415	210	179.51	0.366	570	303.95	0.325
140	43.80	0.413	220	183.17	0.365	580	307.20	0.323
130	47.93	0.411	230	186.82	0.364	590	310.43	0.322
120	52.04	0.409	240	190.46	0.362	+600	313.65	0.321
110	56.13	0.407	250	194.08	0.362	610	316.86	0.319
-100	60.20	0.405	260	197.70	0.360	620	320.05	0.319
90	64.25	0.403	270	201.30	0.358	630	323.24	0.317
80	68.28	0.401	280	204.88	0.358	640	326.41	0.316
70	72.29	0.399	290	208.46	0.357	650	329.57	0.315
60	76.28	0.397	+300	212.03	0.355	660	332.72	0.314
50	80.25	0.396	310	215.58	0.355	670	335.86	0.313
40	84.21	0.396	320	219.13	0.353	680	338.99	0.311
30	88.17	0.396	330	222.66	0.352	690	342.10	0.311
20	92.13	0.394	340	226.18	0.351	+700	354.21	0.309
10	96.07	0.393	350	229.69	0.350	710	348.30	0.308
+ 0	100.00	0.390	360	233.19	0.348	720	351.38	0.307
- 10	103.90	0.389	370	236.67	0.348	730	354.45	0.306
20	107.79	0.388	380	240.15	0.346	740	357.51	0.304
30	111.67	0.387	390	243.61	0.345	750	360.55	0.304
40	115.54	0.386	+400	247.06	0.344	760	363.59	0.304
50	119.40	0.384	410	250.50	0.343	770	366.61	0.302
60	123.24	0.383	420	253.93	0.341	780	369.62	0.301
70	127.07	0.382	430	257.34	0.341	790	372.62	0.300
80	130.89	0.381	440	260.75	0.339	+800	375.61	0.299
90	134.70	0.380	450	264.14	0.338	810	378.59	0.298
+100	138.50	0.378	460	267.52	0.337	820	381.55	0.296
110	142.28	0.378	470	270.89	0.336	830	384.50	0.295
120	146.06	0.376	480	274.25	0.335	840	387.45	0.295
130	149.82	0.375	490	277.60	0.333	850	390.38	0.293

Figure 7-20

Resistance vs Temperature (For Platinum RTD's)

In its laboratory form, the platinum RTD is the world standard for temperature measurement from  $-270^{\circ}\text{C}$  to  $+660^{\circ}\text{C}$ . In its industrial version, it is a temperature sensor of unequalled accuracy, sensitivity, and stability. Because of its high electrical output, the platinum RTD furnishes an accurate input to indicators, recorders, data-loggers, controllers, and computers.

An RTD is often used in a bridge-type configuration when precision temperature sensing is required. A schematic of a basic bridge-type temperature sensing network is shown in Figure 7-21. In Figure 7-21A,  $R_1$ ,  $R_2$ , and  $R_3$  are fixed  $100\Omega$  precision resistors.  $R_4$  is a platinum wire RTD whose variable resistance is determined by the temperature of the oven it is sensing. In this case, since oven temperature is  $0^{\circ}\text{C}$ , we can refer back to Figure 7-20 and determine that the resistance of  $R_4$  (RTD) is also  $100\Omega$ .

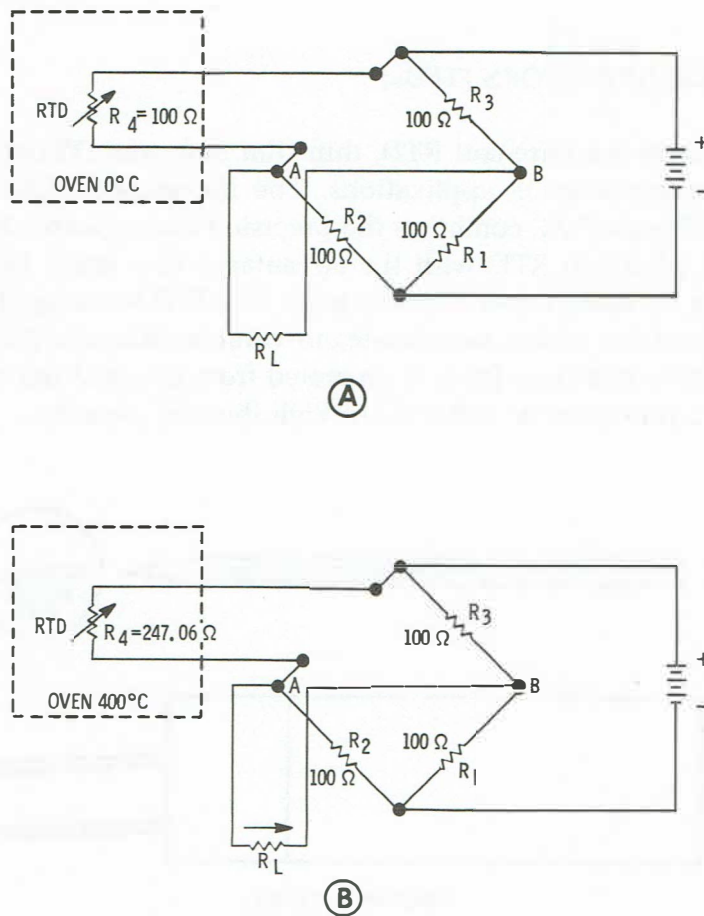


Figure 7-21  
RTD Temperature Sensing Network



Referring to Figure 7-21A, operation of the bridge is based on the fact that, if the voltage at point A is equal to the voltage at point B, no current will flow through resistor  $R_L$ . This “balanced” condition exists any time the ratio of  $R_4/R_2$  equals the ratio of  $R_3/R_1$ . Thus, if  $R_1$  equals  $R_2$ , zero current will flow through  $R_L$  when  $R_4 = R_3$ . Obviously, if  $R_4$  does not equal  $R_3$ , then a potential difference exists between A and B, and some current will flow through  $R_L$ .

If the oven temperature in Figure 7-21B rises to  $+400^\circ\text{C}$ , the resistance of  $R_4$  (RTD) will increase accordingly. Again, referring to Figure 7-20, we see that the resistance of  $R_4$  is now  $247.06\Omega$  which is larger than the fixed resistance of  $R_3$ . Therefore, B will be more positive than A, and current will flow from A through  $R_L$  to B. On the other hand, if oven temperature goes below  $0^\circ\text{C}$ ,  $R_4$  would be less than  $R_3$  and current would flow from B through  $R_L$  to A. As you can see, a small change in oven temperature could easily be “sensed” if a comparator circuit were used in place of  $R_L$ .

### THIN FILM DETECTORS (TFDs)

In addition to the wire-coil RTD, thin film detectors (TFDs) are being used for many sensing applications. The flat-shaped platinum TFD, shown in Figure 7-22, combines the precision temperature characteristic of the platinum RTD with the advantages of a small temperature measuring tip and a rapid response time. The TFD has a rapid response time, since it has a large surface-area-to-volume ratio; also the temperature sensitive platinum layer is separated from the medium to be measured by a thin ceramic substrate of high thermal capacity.

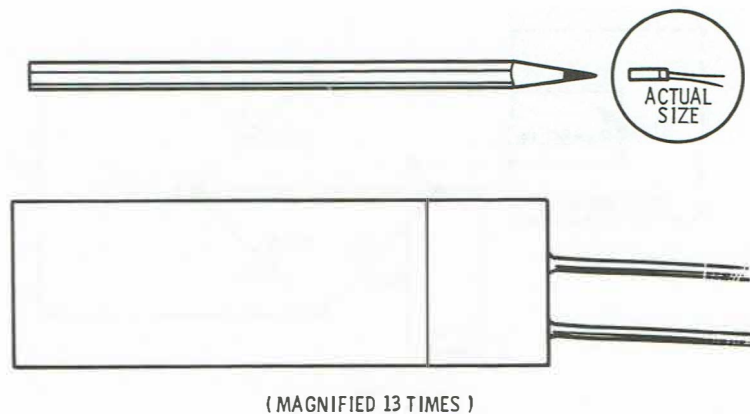


Figure 7-22  
Platinum Thin Film Detector (TFD)

Thin film detectors have a temperature range between  $-200^{\circ}\text{C}$  and  $+500^{\circ}\text{C}$  and a thermal response time of .15 second to attain 63% of temperature change. The TFD has temperature characteristics similar to that of the RTD, shown in Figure 7-20. Like the RTD, the TFD is used in a bridge-type configuration for highly accurate and stable temperature sensing.

Because of their small size and thin, flat packaging, thin film detectors are ideally suited for surface temperature sensing. They may be taped to, cemented onto, or even embedded into the material they are sensing.

## Semiconductor Sensors

Unlike the RTD and TFD, the resistance of a semiconductor decreases as temperature increases. Thus, semiconductors are said to have a negative temperature coefficient.

The most common type of temperature sensing semiconductor, the thermistor, is shown in Figure 7-23. Thermistors exhibit rapid and extremely large changes in resistance for relatively small changes in temperature, as shown by the "temperature vs resistance" curve in Figure 7-24. However, unlike the RTD and TFD, thermistors do not provide a linear response for a change in temperature. Besides their nonlinear response, thermistors have other disadvantages such as low operating currents (usually less than  $100\text{ }\mu\text{A}$ ), limited temperature ranges (between  $-75^{\circ}\text{C}$  and  $+250^{\circ}\text{C}$ ), and the tendency to drift over long periods of use.

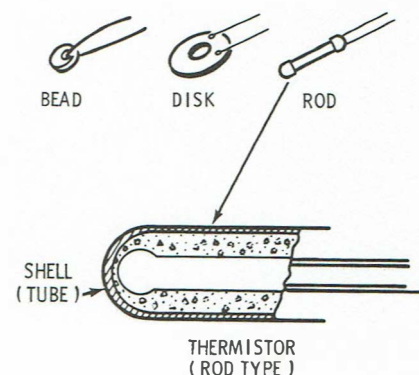


Figure 7-23  
Thermistor Temperature Sensors

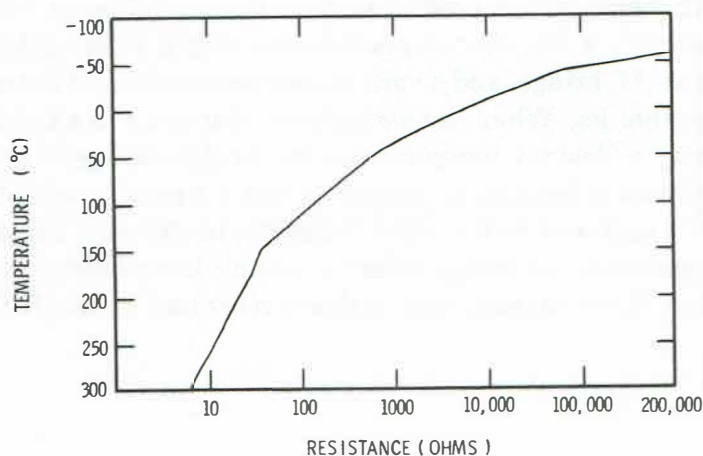


Figure 7-24  
Temperature vs Resistance Curve

Thermistors are available in various sizes and shapes, but they all contain a semiconductor of ceramic material made by sintering mixtures of metallic oxides such as copper, iron, uranium, cobalt, manganese, or nickel. They are then formed into small glass beads, rods, or disks, and because of their small size, are used where other temperature sensors cannot be used.

In addition to thermistors, semiconductor temperature sensors are made from silicon crystals and gallium-arsenide diodes. Silicon-crystal temperature sensors are generally designed as small, very thin wafers and used for surface temperature sensing. Silicon temperature sensors have a temperature-vs-resistance relationship that is similar to the wire-coil RTD between about  $+280^{\circ}\text{C}$  and  $-50^{\circ}\text{C}$ . Below  $-50^{\circ}\text{C}$ , the temperature-vs-resistance relationship is similar to that of a thermistor. Therefore, silicon crystal sensors differ from thermistors in that their temperature coefficient is positive above  $-50^{\circ}\text{C}$ , and their temperature-vs-resistance characteristics are much more linear. This is especially true for their most usable range,  $-50^{\circ}\text{C}$  to  $+250^{\circ}\text{C}$ .

Gallium-arsenide diodes (PN junction) are used for very low temperature sensing. If a constant forward current is maintained through a gallium-arsenide diode, the forward voltage will vary almost linearly with temperature over a range of  $2^{\circ}\text{K}$  to  $70^{\circ}\text{K}$  and  $100^{\circ}\text{K}$  to  $300^{\circ}\text{K}$ . Gallium-arsenide diodes, because of their low cost and durability, are very often used in indoor/outdoor type electronic thermometers. The thermistor, on the other hand, is usually used in a bridge-type circuit, much the same as the RTD and TFD.

Figure 7-25 shows two temperature control circuits using thermistor sensors. The temperature control system, shown in Figure 7-25A, uses a thermistor with a known temperature/resistance relationship to form one leg of an AC bridge, and a variable resistor calibrated in temperature to form another leg. When the variable resistor is set to a specific value, representing a desired temperature, the bridge becomes unbalanced. This unbalance is fed into an amplifier that actuates a switching circuit to provide a source of heat or cold. When the thermistor “senses” the desired temperature, the bridge is balanced and the amplifier receives no input. Thus, the switching circuit deactivates and turns off the heat or cold.

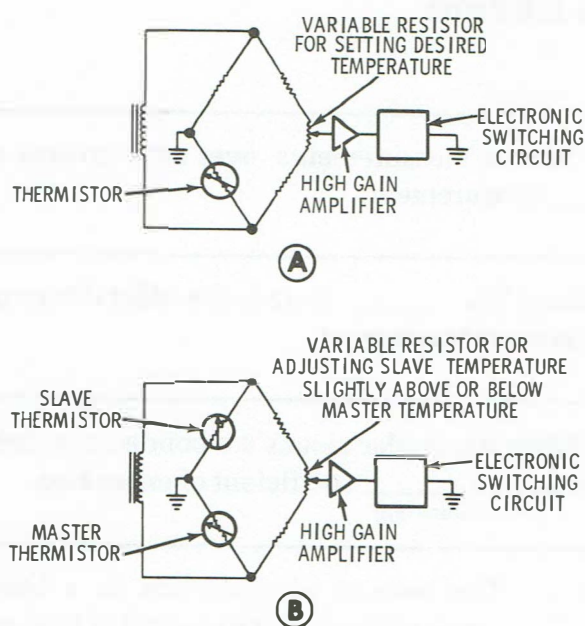


Figure 7-25

(A) Temperature control circuit.

(B) Master-slave control circuit.

The master-slave control system, shown in Figure 7-25B, is used when there is a need to control one temperature with respect to another; for instance, in a process requiring a product to go through a series of baths. The first bath acts as a master and uses a thermistor to sense temperature. Succeeding baths, also using thermistors, are slaves. When these thermistors are placed in the controller bridge, the slave baths can be kept at a temperature relative to the master bath.

The master bath can be controlled with the system described in Figure 7-25A. The master-slave controller can be used to control as many baths as necessary.

## Programmed Review

14.	Of all industrial measurements, over 50% involve some sort of _____ measurement.
15.	(temperature) The _____ scale is the official international unit of temperature measurement.
16.	(Kelvin) When dissimilar metals are bonded together and heated they have an _____ coefficient of expansion. (even/uneven)
17.	(uneven) The amount of deflection in a bimetal-strip is _____ proportional to the amount of heat applied. (directly/inversely)
18.	(directly) In the bimetallic-strip thermostat shown in Figure 7-17A, the amount of heat required to close the circuit can be controlled by adjusting the _____ contact. (stationary/movable)
19.	(stationary) The bimetallic disc switch has a _____ temperature setting at which it will activate. (fixed/variable)
20.	(fixed) Generally, the resistance of a metal conductor _____ as temperature rises. (increases/decreases)
21.	(increases) The platinum RTD has a _____ temperature vs resistance relationship. (linear/nonlinear)
22.	(linear) Because of its high _____ output, the platinum RTD provides accurate inputs to control and recording devices.



23.	(electrical)	The TFD provides a _____ response time due to its face-area-to-volume ratio. (fast/slow)
24.	(fast)	TFDs are ideally suited for _____ temperature sensing applications.
25.	(surface)	The resistance of a semiconductor usually _____ as temperature increases. (increases/decreases)
26.	(decreases)	In general, thermistors produce a _____ response for a change in temperature (linear/nonlinear)
27.	(nonlinear)	Above $-50^{\circ}\text{C}$ , silicon crystal sensors have a _____ temperature coefficient.
28.	(positive)	A bridge circuit is said to be _____ when no output occurs.
	(balanced)	

## ULTRASONIC SENSORS

Ultrasonic sensing systems use a form of sound energy that lies beyond the range of human hearing. That is, they use sound waves whose frequencies are more than 15,000 Hertz or vibrations per second. Ultrasonic sensing systems use a combination of acoustics and electricity. "Acoustics" refers to the science that deals with the generation, control, transmission, reception, and effects of sound passing through gases, liquids, and solids. The electrical portion of ultrasonics deals with the generation of high-frequency electrical energy. To give you a better understanding of ultrasonic sensing systems, we will first explore the properties of sound.

### Properties of Sound

Sound travels from its source to a detector in the form of waves and is considered to be a form of mechanical energy. To generate such waves requires a material medium that contains particles able to interact with each other. This requirement is satisfied by the presence of a gas, liquid, or a solid. Unlike electromagnetic waves, sound waves cannot be transmitted through a vacuum.

When an object vibrates in air, the air molecules next to its surface are alternately contracted and expanded. As these molecules contract and expand, they collide with their neighboring molecules, causing them to also contract and expand. Thus, a series of contractions and expansions are transmitted through the air, ultimately arriving at a detector.

If the material between the vibrating source and the detector is a liquid or solid, the sound is transmitted in a slightly different manner. The molecules in solids and liquids are held together by forces, the magnitude of which depends on the elastic properties of the material. The speed at which these molecules can be expanded and contracted, and therefore interact with their neighbors, is determined by the elasticity of the material. Thus, the velocity at which sound waves travel through a material depends upon its elasticity — the less the elasticity the greater the velocity.

As a rule, sound waves travel faster through both solids and liquids than they do through gases. This is due to the extra time required for collisions to occur between the molecules of a gas (gases are more elastic, and so have a slower velocity). For the same reason, solids are less elastic than liquids; therefore, sound waves travel faster through solids than through liquids. As an example, the velocity of sound in steel is approximately 11,180 mph; in water approximately 3130 mph; and in air at room temperature (68°F) at sea level, the velocity of sound is 769 mph.

The main determining factors affecting the velocity, namely elasticity and density of the material, do not change appreciably with temperature in solids and liquids; thus temperature changes have little effect on the velocity of sound in such materials. In the case of gases, however, temperature changes have a marked effect on velocity, since increasing the temperature causes the gas molecules to move from collision to collision with greater speed. The velocity of sound in air for example increases from 769 mph at 68°F to 864 mph at 212°F.

Sound waves are reflected from surfaces in their path in much the same manner as reflection occurs in optics. That is, when sound travels through a medium, either gas, liquid, or solid, and strikes an interface, a portion of the sound wave is reflected back to the source. (An interface occurs when two different mediums meet, for example, when two solids meet, when a solid meets a liquid, when a solid meets a gas, when a liquid meets a gas, etc. The point at which they meet is called an interface.)

## Ultrasonic Sensing Systems

Ultrasonic sensing systems are used to sense such functions as level, motion, and material thickness, plus others. With additional circuitry, a basic ultrasonic sensing system can be used to navigate or guide a mobile robot.

The two main components of an ultrasonic sensing system are the transmitting and receiving transducers. The ultrasonic transmitting and receiving units can be separately mounted, individually housed in a common assembly, or, they may be combined and housed in a single sensor. The piezoelectric transducer, which can be either a transmitter or a receiver, is the most common type of ultrasonic sensor used.

## PIEZOELECTRIC TRANSDUCERS

Certain natural crystalline substances, such as quartz, tourmaline, and Rochelle salt, have an unusual electrical characteristic. If a static mechanical pressure is applied to the crystal, a DC voltage is generated. Likewise, if a vibrating mechanical pressure is applied, an AC voltage is generated. Conversely, if an AC voltage is applied across the crystal, the crystal undergoes a physical change, resulting in mechanical vibrations. This relationship between electrical and mechanical effects is known as the **piezoelectric effect**.

Because of their structure, crystals have a natural frequency of vibration. If the frequency of the applied AC signal matches this natural frequency, the crystal will vibrate a large amount. However, if the frequency of the exciting voltage is slightly different than the crystal's natural frequency, little vibration is produced. The crystal, therefore, is extremely frequency selective and frequency of vibration is extremely constant, which makes it ideally suited for sensing applications.

The natural frequency of a crystal is usually determined by its thickness. As shown in Figure 7-26, the thinner the crystal is, the higher is its natural frequency. Conversely, the thicker the crystal is, the lower is its natural frequency. To obtain a specific frequency, the crystal slab is ground to the required dimensions. Of course there are practical limits on just how thin a crystal can be cut, without it becoming extremely fragile.



**Figure 7-26**  
Thickness Determines the Crystals Natural Frequency

The piezoelectric crystal transducer, shown in Figure 7-27, can be a transmitter, a receiver, or both, depending upon circuit configuration. In a piezoelectric crystal transmitter, the natural crystal vibration frequency is called the **resonant frequency**. This is the frequency at which the most violent physical changes occur when voltage is applied. If the applied voltage is alternating at an ultrasonic frequency, above 15,000 Hz, the piezoelectric material will contract and expand (vibrate) at the same ultrasonic frequency. These vibrations are then transferred to the metal plates or diaphragm to produce ultrasonic waves.

The receiving piezoelectric crystal is essentially the same as the transmitting crystal. However, its operation is dependent upon the signal sent out by the transmitter. When ultrasonic waves from the transmitter strike the receiving crystal's metal plates it causes them to vibrate. These vibrations are felt by the piezoelectric material which in turn contracts and expands. The contraction and expansion of the piezoelectric material causes a small AC signal to be generated. This small signal is then amplified and used to actuate the control element of the system. If the transmitting crystal and the receiving crystal have the same resonant frequency, maximum vibration of the transmitter will cause maximum vibrations to be felt by the receiver. This in turn, will produce maximum receiver signal output.

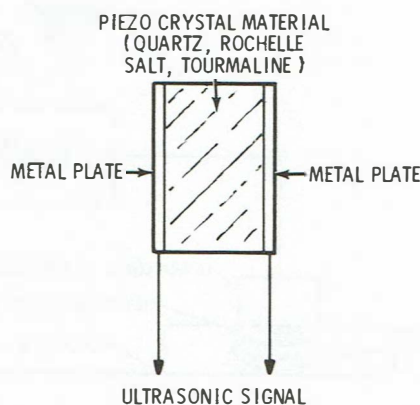


Figure 7-27  
Piezoelectric Transducer



## Ultrasonic Thickness Measuring System

Figure 7-28 shows an ultrasonic sensing system being used for quality control. In this case, ultrasonic vibrations are used to check the thickness of a sheet of steel. To measure the thickness, the transducer is placed on top of the steel sheet so that the vibrations pass through the steel to the interface. As we know, when sound vibrations strike an interface, a portion of the vibrations will be reflected back to the transducer. Thus, the time it takes for the ultrasonic vibrations to travel through the steel, meet the interface, and be reflected back through the steel depends, among other factors, on the thickness of the steel.

If the thickness of the steel is such that the time required for one down and back trip of ultrasonic energy is equal in time to one cycle of ultrasonic vibration, a condition of resonance is produced. Resonance, as you recall, occurs when vibrations return at the exact time to aid outgoing vibrations. Therefore, resonance would depend on the thickness of the steel and oscillator frequency. Resonance will cause a sudden change in the load that the transducer offers the oscillator. This, in turn, will cause a change in the oscillator current (the collector current of a transistor oscillator). By monitoring the oscillator current and noting the frequency of the oscillator where the current change takes place, the time required for a round trip of ultrasonic vibrations can be determined. Thus, the thickness of the steel can be determined.

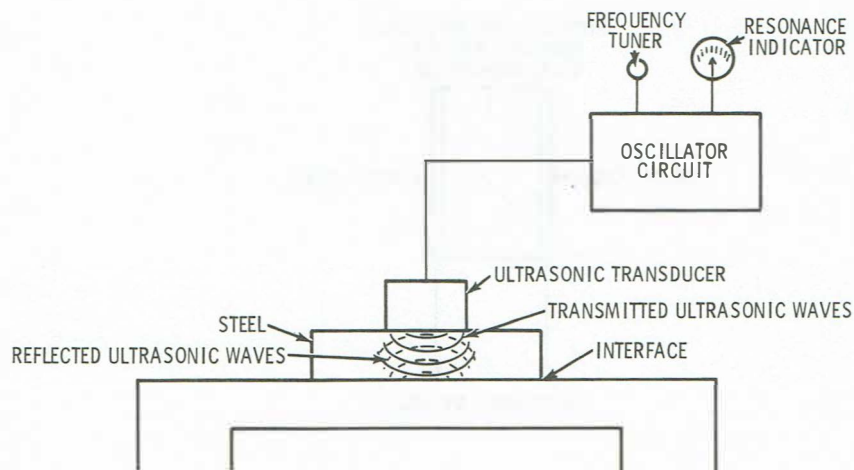


Figure 7-28  
Ultrasonic Thickness Sensor

This type of quality control sensing system is ideal for checking tolerance of parts being manufactured. For instance, an ultrasonic transducer operating at a set frequency, representing a certain thickness of a specific part, could be mounted on a robot's manipulator. The robot could be programmed to move to the part to be inspected and place the transducer on the part. If the part were of the correct thickness, resonance would occur; which in turn would cause a change in oscillator current in the sensing circuit. This change in current could be used to command the robot to pick up the part and place it in the "ACCEPT" bin. Conversely, if the part were not of the correct thickness, resonance in the sensing circuit would not take place; therefore no command would be sent to the robot. After waiting a predetermined amount of time with no input from the sensing circuit, the robot would, due to its program, pick up the part and place it in the "REJECT" bin. The robot would then await the arrival of a new part to be inspected.

## **Ultrasonic Detection and Ranging Navigation**

In order for a mobile robot to move about on its own, it must have the capability of sensing objects in its path. This "object sensing" capability could possibly be accomplished by placing limit switches around the robot. The robot could then travel in a given direction until it bumped into an object, thereby activating the limit switches. Activating the limit switch could cause the robot to back up and seek a new path of travel. This "bump-and-move" method of sensing would not be very practical due to the amount of time required for the robot to accidentally find its way. Besides, it would be rather hard on the robot, and on the objects in its path.

Another, more practical, solution to the problem could be ultrasonic sensing. Ultrasonic sensing is fairly low cost, accurate, and, when used with microprocessor control, relatively easy to accomplish. One possible ultrasonic navigation system, in block diagram form, is shown in Figure 7-29. A more detailed description of the Ultrasonic Detection and Ranging Navigation System (UDARN) will be presented later in the course.

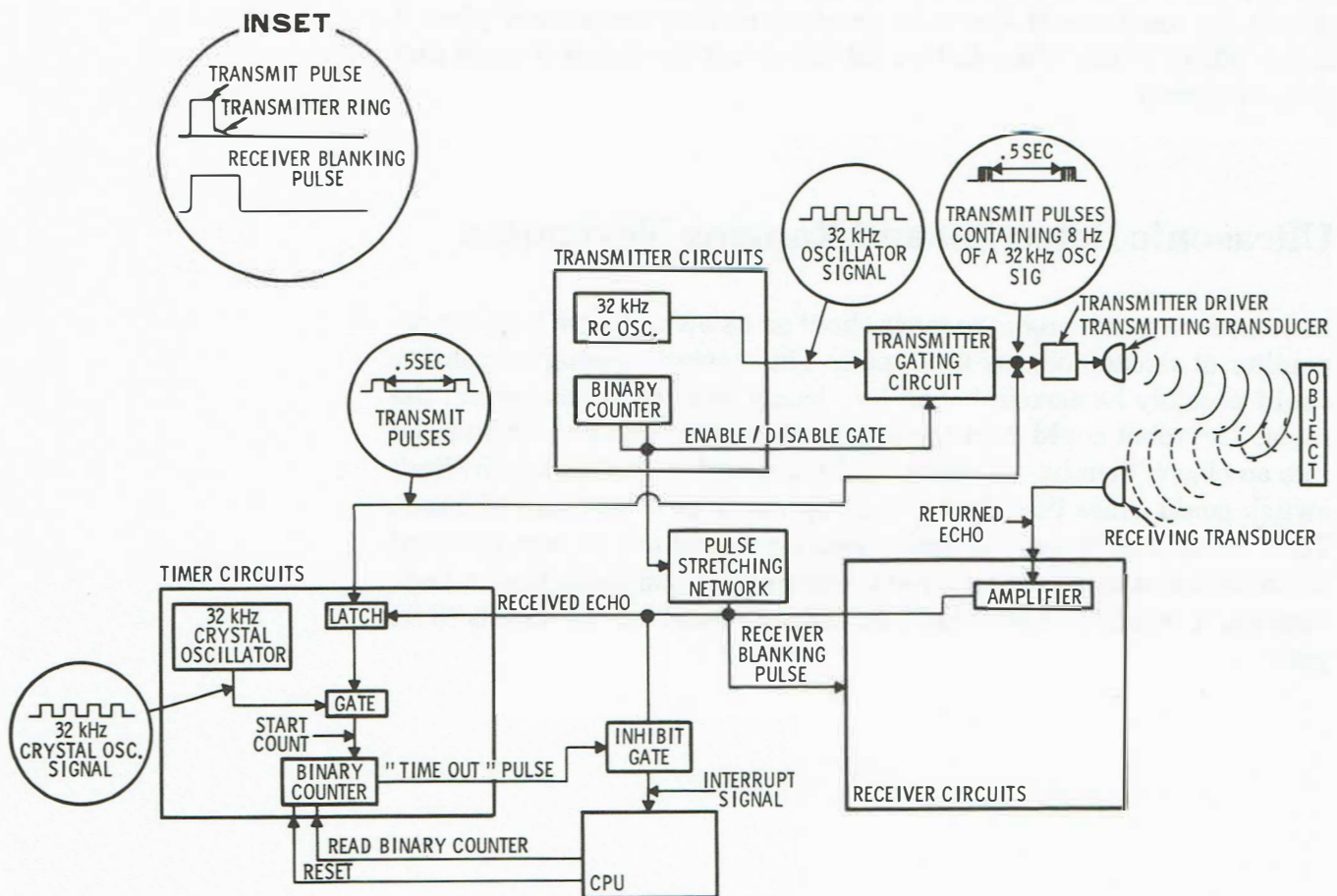


Figure 7-29  
Ultrasonic Detection and Ranging System

## ULTRASONIC DETECTION SYSTEM OPERATION

The ultrasonic detection system consists of one or more transducers (to send and receive ultrasonic waves), electronic circuits for the generation and detection of the ultrasonic waves to and from the transducers, and a timing circuit to control system operation. In Figure 7-29, there is a separate transmitting and receiving transducer. For ease of explanation, we will consider the transmitting and receiving transducers to be stationary; however, in actuality, they would be mounted on a rotating platform to provide 360° coverage.

The ultrasonic transmitting circuit generates 8 cycles of a 32kHz signal at .5-second intervals to drive the transmit transducer. In addition, receiver blanking and timing signals are also produced by the circuit.

The transmitting circuit contains an RC oscillator which is set to oscillate at approximately 32kHz. The output of the oscillator is fed to a transmitter gating circuit, which, when enabled, will permit 8 cycles of the 32kHz signal to be applied to the transmitter drive circuit. This circuit will cause the transmitting transducer to output ultrasonic waves at a corresponding rate.

A binary counting circuit is used for overall transmitter timing. Assume that the counter has been running for some time and the most significant bit is about to go high. This high will reset the entire counter, and at the same time enable the transmitter gating circuit. After 8 cycles of the 32kHz signal has passed through the gating circuit, an output from the binary counter will disable the gating circuit. The transmitter gating circuit will remain disabled until the binary counter once again reaches its maximum count; thus starting the cycle once again.

A portion of the enabling signal going to the transmitter gating circuit is also applied to a pulse stretching network. Here, the signal is stretched to produce a blanking signal for the receiver. The stretched signal is of sufficient length to blank the receiver during transmit and to also blank out any transmitted pulse ringing. See Figure 7-29 inset.



The timing circuit measures the time interval between the leading edge of the transmitted pulse and the leading edge of the received echo. This interval is a measure of the range to the object causing the echo. Only a short range (approximately 10 feet) is required for navigation. So, if no echo pulse is received within approximately 20 milliseconds after the transmit pulse (which represents a range of approximately 11 feet) the timer circuit resets itself and awaits the next transmit pulse to begin a new cycle of operation.

The timer circuit contains a crystal oscillator which provides precise 32kHz clock pulses, through a latch-controlled gating circuit, to a binary counting circuit. When a pulse corresponding in time to the transmit pulse is applied to the latching circuit, the gating circuit becomes enabled; thus allowing the 32kHz signal to be applied to the binary counting circuit, causing it to count. This begins the time interval measurement.

When an echo is received from an object within approximately 10 feet of the robot, the binary counter stops counting and holds its present count. This is accomplished as follows: The received pulse is amplified by the receiver circuitry and applied to the reset input of the binary counter gating latch. The reset causes the gate to become disabled, which in turn, prevents the 32kHz pulses from reaching the binary counter. Therefore, the counter will stop counting and hold its count.

The received echo is also used to generate an interrupt to the CPU. The CPU services the interrupt by reading the binary counter. After reading the binary counter, the CPU generates a reset pulse, thereby readying the circuit for another cycle of operation. If the closest object is more than about 10 feet from the robot, the binary counter will “time out” and inhibit the interrupt signal to the CPU as follows: The binary counter will continue counting until it reaches a count that is equal to approximately 10 feet of ultrasonic range. Once it reaches the required count, it will produce an output that will inhibit any received echo from creating an interrupt for the CPU. In this case the circuit is reset by the next transmit pulse as already described.



## Programmed Review

29.	Ultrasonic sensing systems produce sound waves whose frequencies are more than _____ Hertz or vibrations per second.
30.	(15,000) Sound waves are considered to be a form of _____ energy.
31.	(mechanical) Sound waves cannot be transmitted through a _____.
32.	(vacuum) Sound waves travel faster through _____ than they do through liquids.
33.	(solids) The velocity of sound waves traveling through air _____ as temperature decreases. (increases/decreases)
34.	(decreases) When sound waves strike an _____, a portion of the wave is reflected back to the source.
35.	(interface) If a vibrating _____ pressure is applied to a piezoelectric crystal, an AC voltage is generated.
36.	(mechanical) The piezoelectric crystal produces _____ vibrations when an applied AC signal matches its natural frequency. (constant/varying)
37.	(constant) The thickness of a piezoelectric crystal will determine its _____ frequency.
38.	(natural) A transmitting transducer _____ be used as a receiving transducer. (can/cannot)
39.	(can) Resonance in a circuit occurs when reflected energy _____ transmitted energy. (aids/opposes)
	(aids)

## EXPERIMENT

Perform Experiment 13. This experiment can be found in Unit 12. After you have finished the experiment, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct answers that appear after the exam.

1. Many optoelectronic sensing systems use LEDs whose wavelengths are:
  - A. Approximately 500 nanometers long.
  - B. Approximately 900 nanometers long.
  - C. Approximately 600 nanometers long.
  - D. Approximately 750 nanometers long.
2. Which of the following terms is used to describe the energy or intensity level of emitted light?
  - A. Phonemes
  - B. Photons
  - C. Protons
  - D. Phonons
3. If a phototransistor has only two leads, instead of three, protruding from the body, which of the following leads has been omitted?
  - A. The collector lead.
  - B. The emitter lead.
  - C. The base lead.
  - D. All phototransistors must have three leads.

4. If the light intensity to a phototransistor were decreased, which of the following statements would be true?
  - A. Both the base current and the collector current flowing through the transistor would decrease.
  - B. Both the base current and the collector current flowing through the transistor would increase.
  - C. The base current flow through the transistor would increase, and the collector current flow through the transistor would decrease.
  - D. The base current flow through the transistor would decrease, and the collector current flow through the transistor would increase.
5. When comparing a photo-Darlington detector to a phototransistor detector, which of the following is **not** a characteristic of the photo-Darlington detector?
  - A. The photo-Darlington is more sensitive.
  - B. The photo-Darlington produces a higher output current.
  - C. The photo-Darlington is interconnected with a bipolar transistor and placed in a single package.
  - D. The photo-Darlington responds faster to a change in light intensity.
6. Which of the following sensing operations is **not** normally accomplished using an interrupter module.
  - A. Speed sensing.
  - B. Thickness of a semi-transparent material.
  - C. Vibration sensing.
  - D. They can all be accomplished using interrupter modules.
7. A temperature of 30°C would correspond to which of the following temperatures?
  - A. 49°F.
  - B. 22°F.
  - C. 283.2°K.
  - D. 545.7°R.

8. Which of the following RTDs would provide the most accurate output?
  - A. A copper coil RTD.
  - B. A platinum coil RTD.
  - C. A brass coil RTD.
  - D. A nickel coil RTD.
9. In regards to a nickel coil RTD, which of the following statements is true?
  - A. Coil resistance increases as temperature increases.
  - B. Coil resistance decreases as temperature increases.
  - C. It has a positive temperature coefficient.
  - D. Both A and C are correct.
10. When extreme sensing accuracy is required, most RTDs and TFDs are used in a:
  - A. Bridge configured circuit.
  - B. Delta configured circuit.
  - C. Wye configured circuit.
  - D. Delta-Wye configured circuit.
11. Semiconductor sensors are generally considered to have:
  - A. A wider sensing range than RTDs.
  - B. A greater accuracy than RTDs.
  - C. A negative temperature coefficient.
  - D. A very linear temperature-vs-resistance curve.
12. Which of the following semiconductor sensors has the most linear output between  $-50^{\circ}\text{C}$  and  $-250^{\circ}\text{C}$ .
  - A. Silicon crystal diode.
  - B. Gallium-arsenide diode.
  - C. Ceramic thermistor.
  - D. Platinum wire RTD.



13. If you were to measure a temperature of  $15^{\circ}\text{R}$ , which of the following devices would be best suited for this application?
- A. Silicon crystal diode.
  - B. Gallium-arsenide diode.
  - C. Ceramic thermistor.
  - D. They would all work equally well.
14. Sound waves travel fastest through which of the following mediums?
- A. Air.
  - B. A vacuum.
  - C. Water.
  - D. Steel.
15. The speed at which sound travels through a solid material depends on the materials':
- A. Temperature.
  - B. Thickness.
  - C. Length.
  - D. Elasticity.
16. Ultrasonic waves are considered to be a form of:
- A. Electrical energy.
  - B. Electromagnetic energy.
  - C. Mechanical energy.
  - D. Electromechanical energy.
17. If two steel plates were laid one on top of the other, the area where they joined is called the:
- A. Interface.
  - B. Medium.
  - C. Sound junction.
  - D. Sound gap.

18. If a static mechanical pressure is applied to a piezoelectric crystal, the crystal will:
  - A. Produce an AC voltage.
  - B. Produce a DC voltage.
  - C. Produce violent mechanical vibrations.
  - D. Break into small pieces.
  
19. If a 35kHz signal were applied to a 34kHz piezoelectric crystal, which of the following would take place?
  - A. The crystal would not vibrate.
  - B. The crystal would vibrate at random intervals.
  - C. The crystal would vibrate violently.
  - D. The crystal would vibrate slightly.
  
20. With voltage applied, the frequency at which a piezoelectric crystal most violently vibrates is called:
  - A. Its resonance frequency.
  - B. Its natural frequency.
  - C. Its resonant frequency.
  - D. Both B and C are correct.

14. The following information is for the year ended December 31, 2014:

- a. Sales revenue, \$1,000,000
- b. Sales discounts, \$20,000
- c. Sales returns and allowances, \$10,000
- d. Freight-in, \$5,000
- e. Freight-out, \$3,000

15. The following information is for the year ended December 31, 2014:

- a. Sales revenue, \$1,000,000
- b. Sales discounts, \$20,000
- c. Sales returns and allowances, \$10,000
- d. Freight-in, \$5,000
- e. Freight-out, \$3,000

16. The following information is for the year ended December 31, 2014:

- a. Sales revenue, \$1,000,000
- b. Sales discounts, \$20,000
- c. Sales returns and allowances, \$10,000
- d. Freight-in, \$5,000
- e. Freight-out, \$3,000

## EXAMINATION ANSWERS

For your convenience, the page where the correct answer can be found is shown following the answer.

1. B — Approximately 900 nanometers long. [7-8]
2. B — Photons. [7-7]
3. C — The base lead. [7-11]
4. A — Both the base current and the collector current flowing through the transistor would decrease [7-12]
5. D — The photo-Darlington does not respond faster to a change in light intensity. [7-15]
6. C — Vibration sensing. [7-18,19]
7. D — 545.7 R. [7-25]
8. B — A platinum coil RTD. [7-30]
9. D — Both A and C are correct. [7-29]
10. A — Bridge type circuit. [7-31]
11. C — A negative temperature coefficient. [7-33]
12. A — Silicon crystal diode. [7-34]
13. B — Gallium-arsenide diode. [7-25,34]
14. D — Steel. [7-39]
15. D — Elasticity. [7-38]
16. C — Mechanical energy. [7-38]
17. A — Interface. [7-39]
18. B — Produce a DC voltage. [7-40]
19. D — The crystal would vibrate slightly. [7-40]
20. D — Both B and C are correct. [7-41]

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE

THE HISTORY OF THE



*Unit 8*

**DATA HANDLING  
AND CONVERSION**

## CONTENTS

Introduction .....	8-3
Unit Objectives .....	8-4
Unit Activity Guide .....	8-5
Synchros And Servomechanisms .....	8-6
Digital-To-Analog Converters .....	8-40
Analog-To-Digital Converters .....	8-55
Experiment .....	8-66
Unit Examination .....	8-67
Unit Examination Answers .....	8-71

## INTRODUCTION

In Unit Seven you studied how various sensing devices were used to obtain data. This data would be virtually useless for control purposes if there were no methods available to convert it into a usable format. Control data, as you know, comes in two basic forms — analog and digital. Data is also controlled and used by the same two methods — analog and digital.

Much of the information used in industrial control and robotics is obtained in analog form and used or displayed in the same analog form. Synchro systems and servomechanisms are one of the primary methods of acquiring analog information, converting it to electrical information, and then using the electrical signals to produce an analog display or control function. In the first half of this unit, you will learn how basic synchro systems and servomechanisms are used to display and control analog functions.

Since most modern industrial processing and virtually all medium and high-technology robots use minicomputer or microprocessor controllers, all analog functions must be converted into digital format before the controller can use them. In addition, there are several control functions that can be accomplished only by analog means; therefore, the controller must be able to communicate with these devices in analog format. Hence, we also need methods of converting digital data to analog data. Thus, in the second half of this unit, you will learn how to perform digital-to-analog and analog-to-digital conversions.

Now, examine the “Unit Objectives” on the following page to see what you will learn in this unit. Then follow the instructions in the “Unit Activity Guide” to be sure you perform all of the steps necessary to complete this lesson successfully. Check off each step as you complete it and, in the spaces provided, keep track of the time you spend on each activity.

Portions of this Unit were excerpted from the Heath/Zenith “Microprocessor Interfacing Course” EE-3402, written by Professor Andrew C. Staugaard, Jr.

## UNIT OBJECTIVES

When you have completed this unit, you should be able to:

1. Explain the operation of the following synchro units:
  - Synchro Transmitter.
  - Synchro Receiver.
  - Differential Synchro Transmitter.
  - Differential Synchro Receiver.
  - Control Transformer.
2. Explain the operation of various simple synchro systems.
3. Given the applied voltage, rotor position, and type of synchro unit, determine the amount of voltage induced in various windings of the synchro device.
4. State the difference between a balanced and unbalanced synchro system.
5. Explain the operation of a basic open-loop servomechanism.
6. Explain the operation of a basic closed-loop servomechanism.
7. Describe the operation of a followup control transformer.
8. Explain two methods of providing digital-to-analog conversion.
9. Describe how to interface digital-to-analog converters to a micro-computer via a PIA.
10. Explain three methods of providing analog-to-digital conversion.
11. Describe how to interface analog-to-digital converters to a micro-computer via a PIA.
12. Define the terms resolution, settling time, and accuracy with respect to DACs.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "Synchro Systems."	_____
<input type="checkbox"/> Answer Programmed Review Questions 1-6.	_____
<input type="checkbox"/> Read "Synchro System Operation."	_____
<input type="checkbox"/> Answer Programmed Review Questions 7-18.	_____
<input type="checkbox"/> Read "Control Transformer (CT)."	_____
<input type="checkbox"/> Answer Programmed Review Questions 19-24.	_____
<input type="checkbox"/> Read "Servomechanisms."	_____
<input type="checkbox"/> Answer Programmed Review Questions 25-33.	_____
<input type="checkbox"/> Read "Digital-To-Analog Converters."	_____
<input type="checkbox"/> Answer Programmed Review Questions 34-42.	_____
<input type="checkbox"/> Read "Analog-To-Digital Converters."	_____
<input type="checkbox"/> Answer Programmed Review Questions 43-51.	_____
<input type="checkbox"/> Perform Experiment 14.	_____
<input type="checkbox"/> Complete The Unit Examination.	_____
<input type="checkbox"/> Check The Examination Answers.	_____



## SYNCHROS AND SERVOMECHANISMS

In many instances, the angular position of one quantity is controlled by the angular position of another quantity. When the two quantities are physically close together, control can be accomplished directly, as shown in Figure 8-1A, using shafts, gears, or some other mechanical means. However, when the controlled quantity is some distance away from the controlling quantity, it is usually impractical to interconnect the two by mechanical means. Consequently, some other method must be used to transmit the angular information. The synchro system, shown in Figure 8-1B, or the servo mechanism, which will be discussed later, are two possibilities. These two systems are used to transmit mechanical shaft angles over long distances by means of electrical voltages.

Many times the terms synchro and servo are considered to be synonymous; however, this is not the case. A synchro system transmits mechanical shaft position without power amplification. That is, mechanical power output is equal to mechanical power input, disregarding losses. In applications where the controlling quantity must be amplified in order to provide a large torque to the controlled quantity, the system used is known as a servomechanism. Synchro devices are widely used as basic components of a servomechanism; therefore, we will discuss the synchro system first.

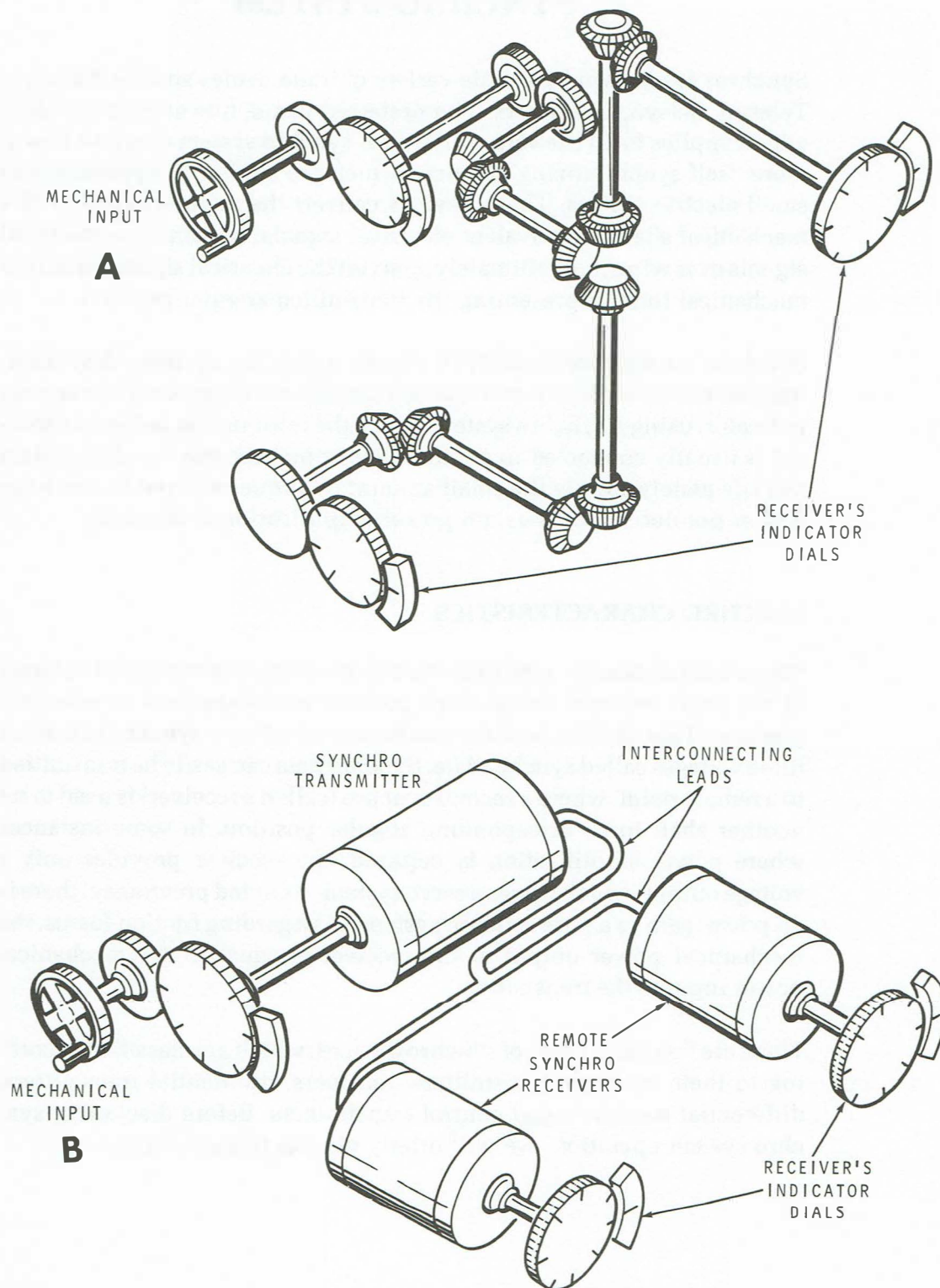


Figure 8-1  
Methods of transferring data:  
(A) mechanical data transfer;  
(B) electrical data transfer using a synchro system.

## SYNCHRO SYSTEM

Synchros are known by a wide variety of trade names such as Autosyn, Telesyn, Selsyn, and others. The preferred name, however, is synchro, which applies to all the various types. A synchro system employs two or more “self synchronizing” devices which are similar in appearance to small electric motors. These devices convert the angular position of a mechanical shaft to equivalent electrical signals; transmit the electrical signals over wires; and ultimately, convert the electrical signals back into mechanical form, representing the transmitted angular position.

Synchros are used extensively in remote indicating systems. Any information that is displayed on a dial can usually be transmitted to a remote indicator, using a synchro system. Since the information being transmitted is usually connected to another dial or pointer, the synchro system can adequately supply the small amount of torque required to move the dial or pointer shaft. Thus, no power amplification is required.

### SYNCHRO CHARACTERISTICS

When information is in the form of shaft position, it is expressed in terms of the angle between actual shaft position and some zero or reference position. This angular position can be converted by a synchro to a set of three voltages called synchro data. Synchro data can easily be transmitted to a remote point, where a second synchro (called a receiver) is used to set another shaft to a corresponding angular position. In some instances where power amplification is required, the receiver provides only a voltage output, which drives a servo system. As noted previously, there is no power gain in a pure synchro system. Disregarding friction losses, the mechanical power output of the receiver is equal to the mechanical power input to the transmitter.

There are five basic types of synchro devices, which are classified according to their function: transmitters, receivers, differential transmitters, differential receivers, and control transformers. Before discussing synchro system operation, we will briefly discuss these devices.

## SYNCHRO TRANSMITTER (TX)

The synchro transmitter, shown schematically in Figure 8-2, is sometimes referred to as a synchro generator. The rotor (1) consists of a single winding, and the stator (2) is made up of three windings displaced 120°. The rotor is excited by an AC source and is usually coupled directly or indirectly, through gears, to a controlling shaft.

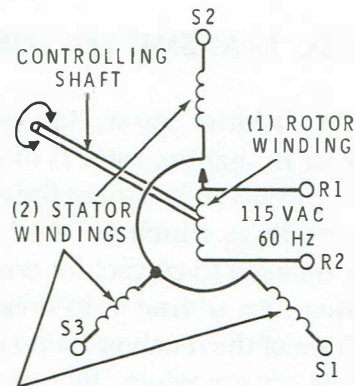


Figure 8-2  
Synchro transmitter (TX) (restrained rotor).

The controlling shaft represents some angular position to be transmitted. This angular position could be representative of a temperature or pressure gauge setting, the position of an indexing table, or even the lateral position of a robot. In some cases, the control shaft is positioned by a person dialing in a specific setting to control some remote device or function. The rotor is usually so restrained that it cannot turn except under the influence of the controlling shaft. The alternating field created by the rotor winding induces voltages in the stator windings that are representative of the angular position of the rotor in respect to the stator windings.

### SYNCHRO RECEIVER (TR)

The synchro receiver, shown in Figure 8-3, is sometimes called a synchro motor. It is similar electrically to the synchro transmitter. However, unlike the synchro transmitter, the rotor of the synchro receiver is free to turn and usually drives a light load, such as a dial, pointer, or some other indicating device. The drive is accomplished directly, or through a light gear train assembly. The angular position that the rotor assumes is dependent upon the stator voltages received from the synchro transmitter.

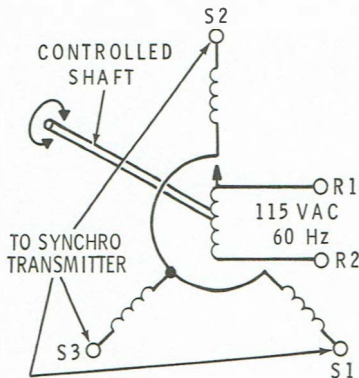


Figure 8-3  
Synchro receiver (TR)  
(free turning rotor).

### DIFFERENTIAL SYNCHRO TRANSMITTER (TDX)

The differential synchro transmitter, shown in Figure 8-4, resembles the regular synchro transmitter in that the rotor is mechanically positioned and the stator is similar. However, the rotor of the differential transmitter contains three separate windings which are electrically displaced 120°. Differential transmitters are used to correct for errors existing in various parts of the synchro system. By adding a differential transmitter to the system, the angular position of the receiver rotor is varied with respect to the transmitter rotor. This occurs when there is a change in the rotor position of the differential transmitter.

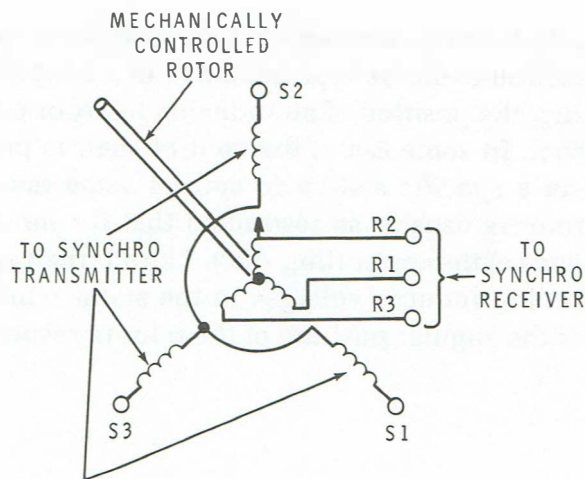


Figure 8-4  
Differential synchro transmitter (TDX)  
(restricted rotor).



### DIFFERENTIAL SYNCHRO RECEIVER (TDR)

The differential synchro receiver, seen in Figure 8-5, is similar in design to the differential transmitter except that the rotor is free to turn. It is used when it becomes necessary to add or subtract two angular quantities. If the differential receiver were connected to two synchro transmitters, its rotor would assume a position corresponding to the angular sum or difference (depending on circuit connections) of the transmitter rotor positions.

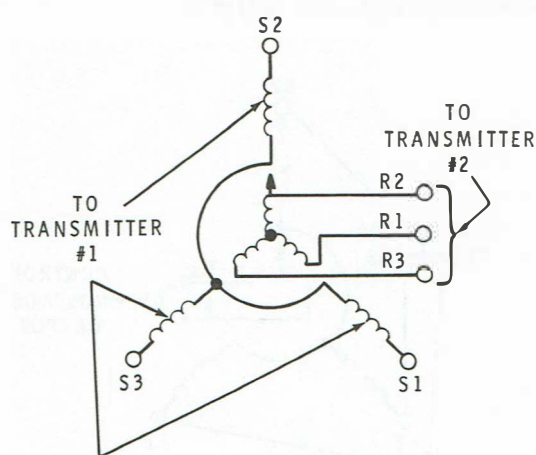


Figure 8-5  
Differential synchro receiver (TDR)  
(free-turning rotor).

### CONTROL TRANSFORMER (CT)

The control transformer, shown in Figure 8-6, is used to indicate angular position when you want to obtain a voltage output only. The control transformer is similar to the synchro receiver, except that the windings of the control transformer have a higher impedance. In addition, the rotor of the control transformer is not free to turn. Therefore, voltages from the synchro transmitter produce a voltage in the rotor of the control transformer, which is representative of the angular displacement of the transmitter rotor.

Now that you have briefly studied the major components utilized in a synchro system, complete the Programmed Review. Then you will study the operation of various synchro systems.

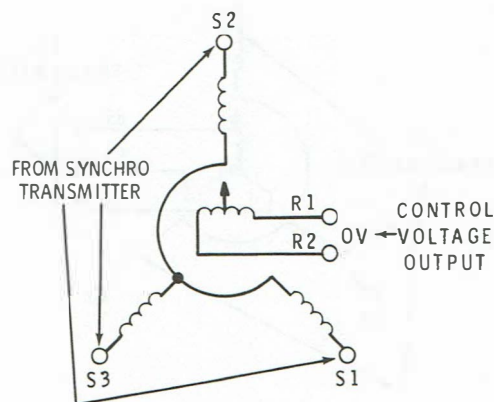


Figure 8-6  
Control transformer (CT) (restricted rotor).

## Programmed Review

1.	A synchro system transmits angular shaft position _____ (with/without) power amplification.
2.	(without) The synchro transmitter has three _____ windings that are electrically displaced 120°.
3.	(stator) In a synchro receiver, the _____ is free to turn and usually drives a light load.
4.	(rotor) A differential _____ is used to correct for errors in a synchro system.
5.	(transmitter) A differential _____ is used to add or subtract the angular inputs from two synchro transmitters.
6.	(receiver) The control transformer produces a _____ output which is representative of the angular displacement of the transmitter rotor.
(voltage)	

## Synchro System Operation

The synchro transmitter operates in much the same manner as a variable transformer, with the rotor acting as the primary winding and the stator windings acting as secondary windings. The voltages induced in each stator winding are proportional to the angle between the rotor and each individual stator winding. Figure 8-7 illustrates a typical synchro system consisting of a transmitter and receiver.

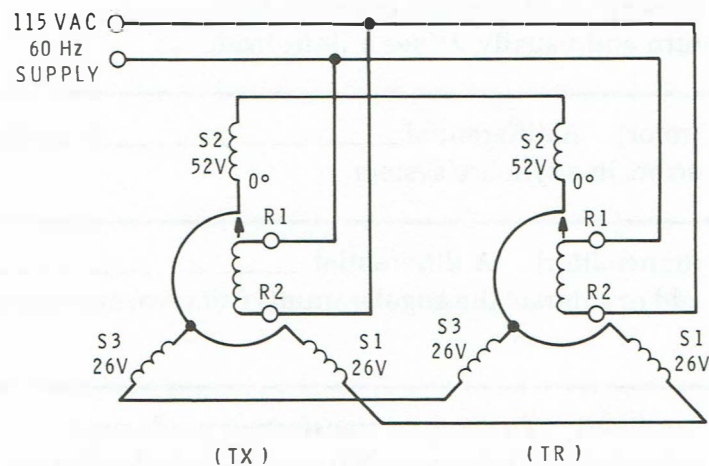


Figure 8-7  
Transmitter-receiver synchro system.

### TRANSMITTER AND RECEIVER SYNCHRO SYSTEM

When an AC voltage, usually 115 VAC 60 Hz, is applied to the rotor (R1, R2) of the synchro transmitter, voltages are induced in the stator windings (S1, S2, and S3). The magnitude and phase of these induced voltages are dependent upon the position of the rotor. Since the turns ratio between the rotor and stator is generally 2.2:1, the maximum voltage that can be induced through transformer action across any individual stator winding is approximately 52 volts.

The rotor of the synchro transmitter, shown in Figure 8-7, is in the 0° position. This 0° position is considered to be the reference position throughout synchro systems. Likewise, designations S1, S2, S3, R1, and R2 are also standard designations for all synchros. With the rotor in the position shown, maximum voltage (52 volts) is induced across stator winding S2.

Stator windings S1 and S3, however, are at 60° angles to the rotor. Therefore, because the voltages induced in each stator winding are proportional to the “cosine” of the angle between the rotor and the individual stator winding, only half as much voltage (26 volts) is induced across S1 and S3.

PROOF: The Cosine Table in Figure 8-8 shows the cosine of 60° to be .500.

Therefore, since 52 volts is the maximum amount of voltage that can be induced in any one stator winding, we find that  $.500 \times 52$  volts is equal to 26 volts.

ANGLE	COSINE	ANGLE	COSINE
0°	1.000	46°	.695
1°	1.000	47°	.682
2°	.999	48°	.669
3°	.999	49°	.656
4°	.998	50°	.643
5°	.996	51°	.629
6°	.995	52°	.616
7°	.993	53°	.602
8°	.990	54°	.588
9°	.988	55°	.574
10°	.985	56°	.559
11°	.982	57°	.545
12°	.978	58°	.530
13°	.974	59°	.515
14°	.970	60°	.500
15°	.966	61°	.485
16°	.961	62°	.470
17°	.956	63°	.454
18°	.951	64°	.438
19°	.946	65°	.423
20°	.940	66°	.407
21°	.934	67°	.391
22°	.927	68°	.375
23°	.921	69°	.358
24°	.914	70°	.342
25°	.906	71°	.326
26°	.899	72°	.309
27°	.891	73°	.292
28°	.883	74°	.276
29°	.875	75°	.259
30°	.866	76°	.242
31°	.857	77°	.225
32°	.848	78°	.208
33°	.839	79°	.191
34°	.829	80°	.174
35°	.819	81°	.156
36°	.809	82°	.139
37°	.799	83°	.122
38°	.788	84°	.105
39°	.777	85°	.087
40°	.766	86°	.070
41°	.755	87°	.052
42°	.743	88°	.035
43°	.731	89°	.018
44°	.719	90°	.000
45°	.707		

Figure 8-8  
Cosine Table.



For ease of explanation, let's assume for a moment that the rotor of the synchro receiver has been removed. Because the stator windings of the receiver are connected directly to the stator windings of the transmitter, the voltages induced in the transmitter stator windings are also applied to the corresponding receiver stator windings. Again referring to Figure 8-7, you see that current flows from S2 of the synchro transmitter through S2 of the receiver and divides equally through S1 and S3, since equal voltages are applied to S1 and S3. Of course, current flow will be reversed when the polarity of the input voltage on the transmitter rotor reverses. This action generates a magnetic field in each receiver stator winding. The important fact to be considered however, is not the magnetic field of each individual stator winding, but rather the resultant magnetic field.

Figure 8-9 illustrates the direction and magnitude of each receiver stator field and the resultant field. As you can see, the resultant synchro receiver field points in a direction corresponding to the position of the transmitter rotor. If you placed an iron bar rotor in the receiver field, it would tend to align itself with the resultant field. However, the iron bar rotor can align itself in either of two positions, 180° apart.

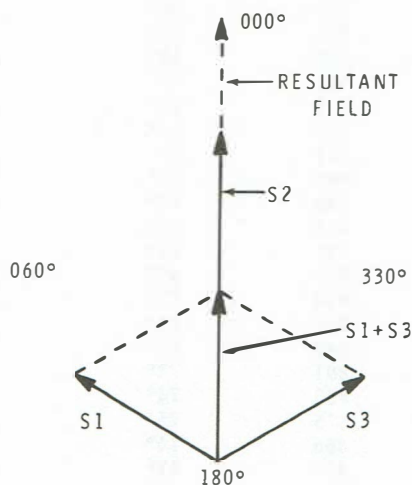


Figure 8-9  
Resultant receiver field  
(transmitter rotor at 0°).

Perhaps you can see this more clearly by observing Figure 8-10. For instance, if the rotor is physically held in position A and then released, it will quickly rotate to position B due to the behavior of magnetic lines of force. Figure 8-10C illustrates the position the rotor will assume if it is rotated to a position where the "Y" end of the rotor is nearer to the electromagnet than the "X" end, and is then released. Thus, you see that a synchro receiver with an iron bar rotor displays a large amount of uncertainty; it can assume two stable positions 180° apart.

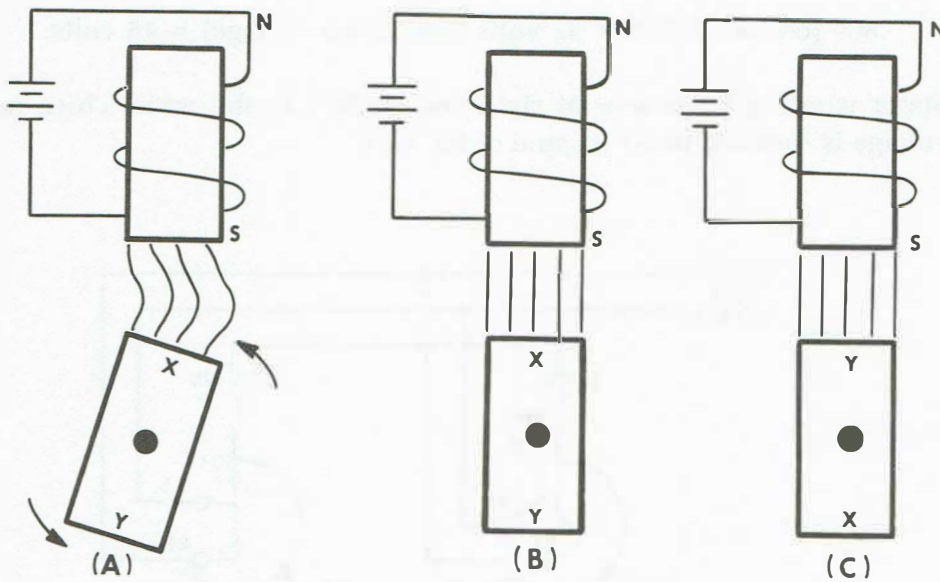


Figure 8-10  
Rotation of an iron bar rotor.

To avoid the possibility of a 180° synchronization error between the transmitter and receiver, the synchro receiver rotor is energized by the same AC source as the synchro transmitter rotor. This makes the receiver rotor an electromagnet. Thus, there is only one possible position it can assume. In addition, the energized rotor provides a much greater torque, and synchro receivers with energized rotors do not have a constant stator current as would an iron bar receiver.

## SYNCHRO TRANSMITTER-RECEIVER OPERATION

Figure 8-11 shows a simple synchro transmitter-receiver system with the transmitter rotor positioned  $30^\circ$  in the clockwise direction from the reference, and the receiver rotor held at  $0^\circ$ . As you recall, voltages are induced in the synchro transmitter stator windings which are proportional to the cosine of the angle between the rotor and the stator windings. With the transmitter rotor now positioned  $30^\circ$  clockwise from the  $0^\circ$  position, stator windings S2 and S3 are displaced  $30^\circ$  from the rotor. As the cosine table in Figure 8-8 shows, the cosine of  $30^\circ$  is .866. Therefore, the voltage induced in S2 and S3 is 45 volts.

$$.866 (\text{cosine of } 30^\circ) \times 52 \text{ volts (maximum voltage)} = 45 \text{ volts.}$$

Stator winding S1 is now at right angles ( $90^\circ$ ) to the rotor. Thus, no voltage is induced in S1 (cosine of  $90^\circ$  is 0).

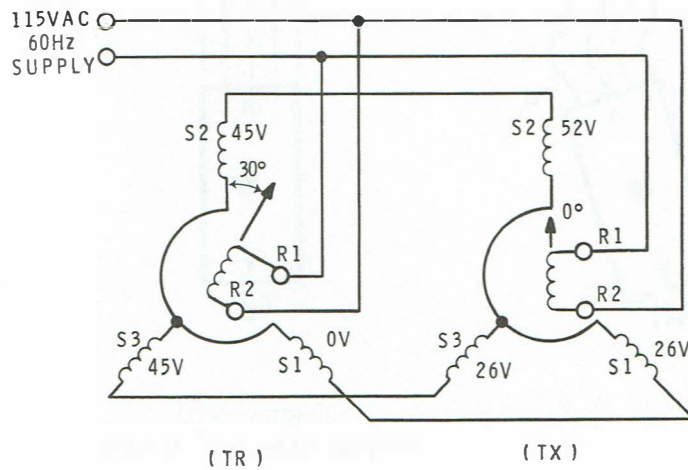


Figure 8-11  
Synchro transmitter-receiver operation  
( $30^\circ$  transmitter offset).

The receiver rotor, which is being held at  $0^\circ$ , induces voltages in the receiver stator windings which are also cosine functions of the angle between the receiver rotor and the receiver stator windings. Receiver stator winding S2 is parallel to the rotor ( $0^\circ$  displacement) and 52 volts is induced in S2.

$$1 (\cosine \text{ of } 0^\circ) \times 52 \text{ volts} = 52 \text{ volts.}$$

S1 and S3 are displaced  $60^\circ$  from the rotor and 26 volts is induced across S1 and S3.

$$.5 (\cosine \text{ of } 60^\circ) \times 52 \text{ volts} = 26 \text{ volts.}$$

The synchro transmitter-receiver circuit is now unbalanced or out of correspondence. Quite simply, voltage differences exist between the stator windings of the transmitter and receiver. These voltage differences cause current to flow within the circuit; and this, in turn, sets up a stator field in the receiver in such a direction as to exert a clockwise torque on the receiver rotor. As you recall, the transmitter is so constructed that its rotor cannot rotate except under the influence of the controlling shaft. However, this is not the case of the receiver rotor, which is free to rotate. Thus, when the receiver rotor (which is being held at  $0^\circ$ ) is released, it rotates in a clockwise direction due to the clockwise torque applied to it.

As the receiver rotor approaches the  $30^\circ$  clockwise offset, produced by the transmitter, the degree of unbalance between the transmitter and receiver decreases. Hence, stator currents decrease since the voltages induced in the receiver stator windings approach the voltages induced in the transmitter stator windings. When the receiver rotor reaches the same position as the transmitter rotor, the voltages induced in the receiver stator windings are equal to the voltages induced in the transmitter stator windings. A balanced or in correspondence condition now exists, and stator currents no longer flow. Therefore, torque is no longer produced and the receiver rotor ceases to rotate.

It can be seen, then, that the transmitter supplies stator current to establish a field in the receiver which produces torque only when the receiver rotor is out of correspondence with the transmitter rotor. Torque produced in the receiver is proportional to the amount of error between the transmitter rotor and receiver rotor. At very small error angles, the torque produced in the receiver may not be sufficient to overcome the friction of the bearings and load. For this reason, friction and load is kept as low as possible in the synchro receiver. The maximum error in transmitter-receiver systems is generally less than  $1^\circ$ .

### SYNCHRO TRANSMITTER — DIFFERENTIAL TRANSMITTER — RECEIVER SYSTEM

Another important and frequently used synchro unit is the differential transmitter. It is used in many circumstances where a correction must be inserted in the angular information being transmitted, or where the sum or difference of two angular quantities must be transmitted. As we stated previously, synchros operate as variable transformers. This statement applies equally to the differential transmitter as well as all synchro units. In the synchro differential transmitter, however, the stator acts as the primary of the variable transformer and the rotor windings serve as the secondary. Figure 8-12 is a schematic diagram illustrating a differential transmitter inserted between a synchro transmitter and receiver.

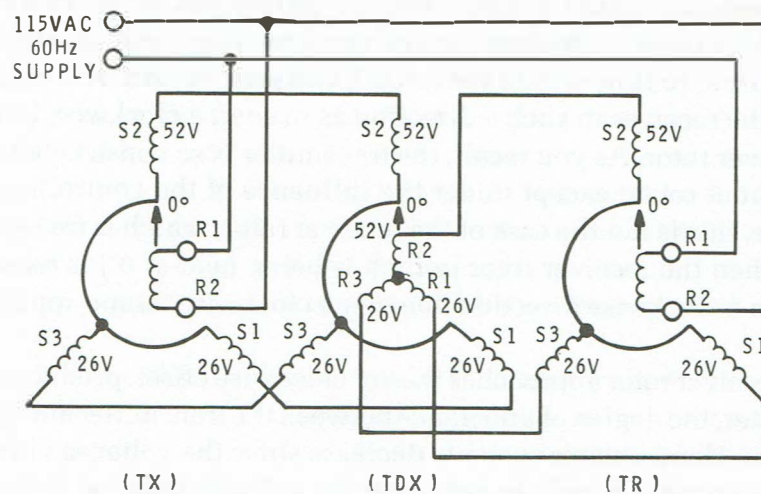


Figure 8-12

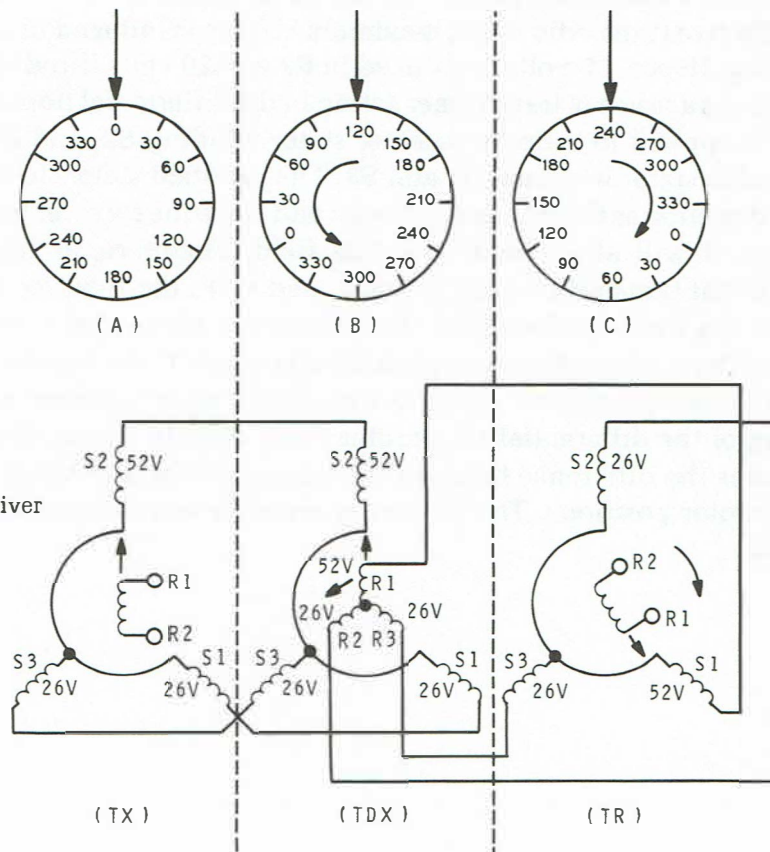
Synchro transmitter — differential transmitter — receiver system (all units at reference position).



In a differential transmitter, unlike the synchro transmitter, the turns ratio between the rotor and stator windings is 1:1; therefore, the voltage ratio is also one to one. With the differential transmitter at its electrical zero ( $0^\circ$ ), synchro data is passed from the transmitter to the receiver without being changed. Referring to Figure 8-12, you see that both the synchro transmitter and differential transmitter are mechanically positioned to  $0^\circ$ . Since the rotor of the synchro transmitter is at  $0^\circ$ , 52 volts is induced into stator winding S2 and 26 volts is induced into stator windings S1 and S3. The stator windings of the synchro transmitter are connected directly to the stator windings of the differential transmitter. Therefore, the voltages induced across the transmitter stator windings are developed across the differential transmitter stator windings. The differential transmitter rotor windings are, in turn, connected to the stator windings of the synchro receiver. Therefore, the voltages that are induced in the differential transmitter rotor windings by transformer action from the stator windings are applied to the stator windings of the synchro receiver.

With the differential transmitter rotor positioned as shown, all three rotor windings, which are electrically  $120^\circ$  apart, form an angle of  $0^\circ$  with their respective stator windings (R1 with S1, R2 and S2, and R3 with S3). Since the effective turns ratio is 1:1, maximum voltage is induced in each rotor winding. Hence, 52 volts is induced in R2 and 26 volts is induced in R1 and R3. As a result of transformer action in the differential transmitter, 52 volts is applied to synchro receiver stator winding S2, and 26 volts is applied to stator windings S1 and S3. The resultant stator field is in the same direction as the transmitter rotor; and since the receiver rotor is free to turn, it will align itself with this field. Therefore, as long as the differential transmitter rotor is positioned at  $0^\circ$ , the receiver rotor will follow the field produced by the transmitter rotor. If the differential transmitter rotor is set to some position other than  $0^\circ$ , the synchro receiver will indicate a position equal to transmitter rotor position minus the setting of the differential transmitter rotor. Simply stated, the receiver indicates the difference between the transmitter and differential transmitter rotor positions. The following example will illustrate this more clearly.

Assume for a moment that all units of the synchro system are set to their respective  $0^\circ$  or reference position. That is, the transmitter R1 and S2 windings are aligned, the differential transmitter R2 and S2 windings are aligned, and the synchro receiver R1 and S2 windings are aligned. Now let us further assume that the synchro transmitter remains at its  $0^\circ$  or reference position, as shown on dial A in Figure 8-13; but now a  $120^\circ$  counterclockwise error has been introduced into the rotor of the differential transmitter. You can see this by observing dial B. Figure 8-13 shows you that maximum voltage (52 volts) is now induced in differential transmitter rotor winding R1, which is connected to stator winding S1 of the receiver. Twenty-six volts is induced into differential transmitter windings R2 and R3, and is applied to the respective receiver stator windings. The resultant receiver stator field will cause the receiver rotor to rotate  $120^\circ$  in the clockwise direction, which will correspond to the  $240^\circ$  position shown on dial C. Thus, we have a  $0^\circ$  or  $360^\circ$  setting on the synchro transmitter minus a  $120^\circ$  error introduced in the differential transmitter equaling the  $240^\circ$  shown on the synchro receiver. In all cases where the differential transmitter is directly connected between the transmitter and receiver, the receiver shaft position is equal to the position of the transmitter shaft minus the position of the differential shaft.



**Figure 8-13**  
Transmitter — differential transmitter — receiver  
synchro system  
(connected for subtraction).

If it is desirable to have the differential transmitter shaft position added to the transmitter shaft position, you simply reverse S1-S3 and R1 and R3 of the differential transmitter. A differential transmitter connected for addition is shown in Figure 8-14. Also, if only the transmitter leads were reversed, S1-S3, you would have an output:  $TX + TDX = -TR$ . If only the receiver connections S1-S3 are reversed, the output becomes:  $TX - TDX = -TR$ . As you can see, the differential transmitter is a very versatile device.

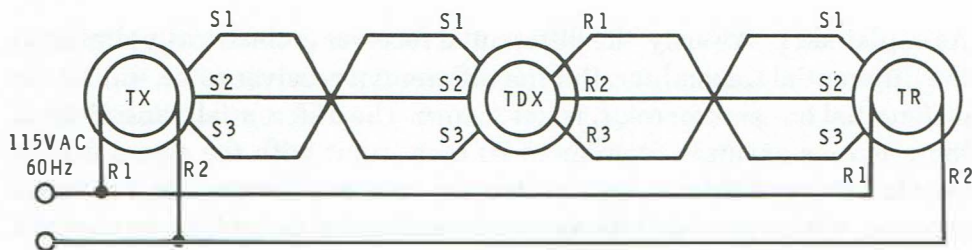


Figure 8-14

Transmitter — differential transmitter — receiver synchro system (connected for addition).

### TRANSMITTER — DIFFERENTIAL RECEIVER — TRANSMITTER SYSTEM

A differential synchro receiver is inserted between two synchro transmitters to indicate the sum or difference of their respective shaft angle positions. If there is no angular difference between the shafts of the two transmitters, the shaft of the differential receiver will be at the zero position. However, if the transmitters are positioned at two different angles, then the differential receiver will indicate either the sum or the difference between them.

As explained previously, the differential receiver is electrically similar to the differential transmitter. But the differential receiver rotor, unlike the differential transmitter rotor, is free to turn. The differential transmitter of the previous examples combined its own input with the signal from a synchro transmitter and transmitted the sum or difference to a synchro receiver, which provided the system's mechanical output. In the case of a differential receiver, the differential unit itself provides the system's mechanical output, usually the sum or difference of the electrical signals received from the two synchro transmitters.

The TX-TDR-TX synchro system shown in Figure 8-15 is connected to produce a difference output from the TDR. The rotor of synchro transmitter number 1 is mechanically positioned to a dial setting of  $75^\circ$  (A), and the rotor of synchro transmitter number 2 is positioned to a dial setting of  $30^\circ$  (C). The voltages induced in the stator windings of transmitter number 1 are applied to the stator windings of the differential receiver; while the voltages induced in the stator windings of transmitter number 2 are applied to the rotor windings of the differential receiver. The voltages applied to the differential receiver produce a resultant field which causes the rotor to rotate to the  $45^\circ$  position, shown on dial (B).

In a simple transmitter-receiver synchro system, a balanced condition is achieved when the voltages applied by the transmitter to the receiver stator windings are canceled by equal and opposing voltages induced across the receiver stator windings by the receiver rotor. The transmitter — differential receiver — transmitter synchro system, shown in Figure 8-15, attains a balanced condition when the differential receiver assumes an angular position equal to the difference between the angular positions of the two transmitters. If either, or both, of the transmitter shaft positions change, the system becomes unbalanced and stator current flows. When stator current flows, it creates a torque that causes the differential receiver to rotate until once again a balanced condition is reached.



If your application requires the sum of two transmitter shaft positions, you simply reverse the R1-R3 leads of the differential receiver.

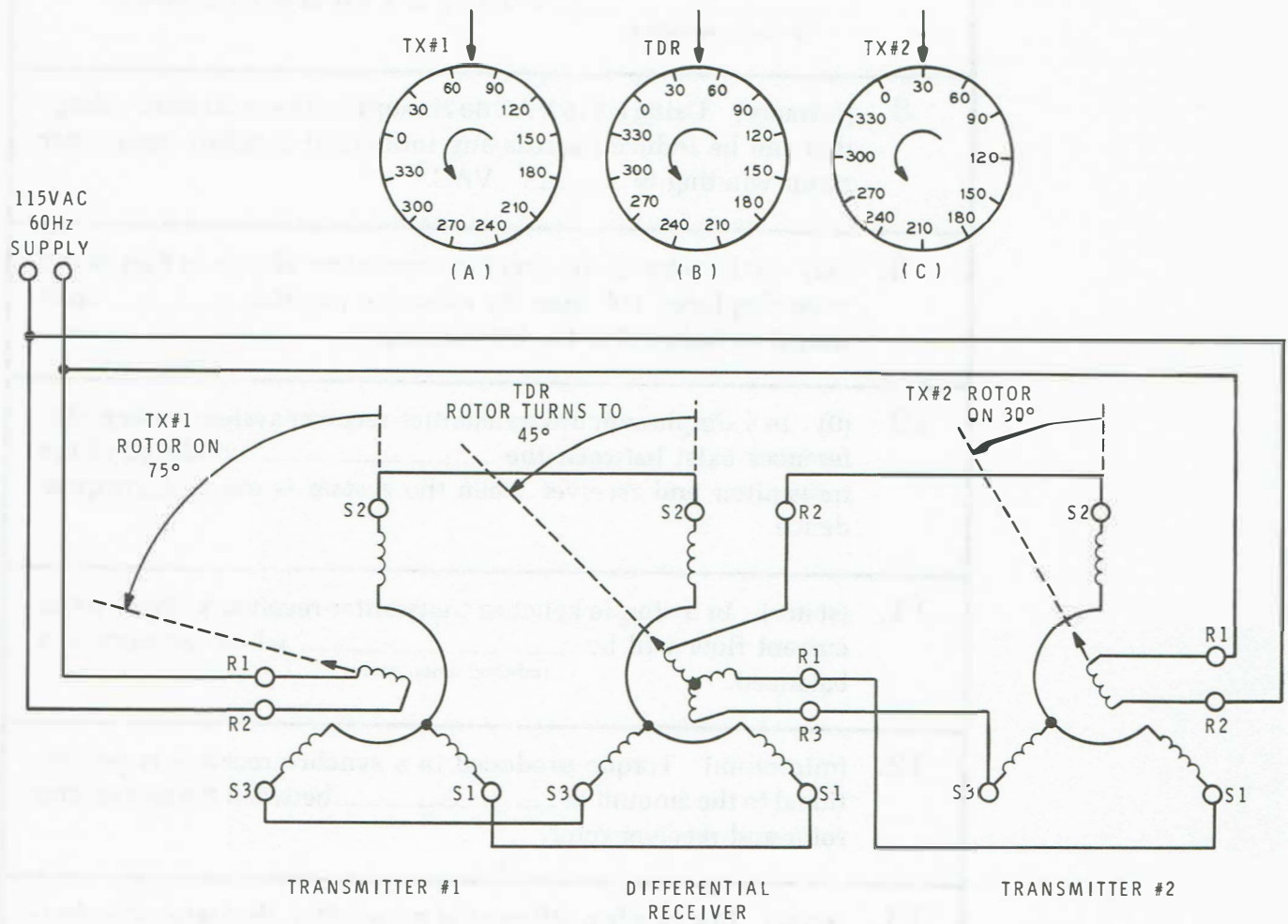


Figure 8-15

Transmitter — differential receiver — transmitter synchro system  
(connected for subtraction).



## Programmed Review

7. The rotor of a synchro transmitter acts similar to the \_\_\_\_\_ winding of a variable transformer.  
(primary/secondary)

8. (primary) Using a 115 VAC 60 Hz supply, the maximum voltage that can be induced across any individual synchro transmitter stator winding is \_\_\_\_\_ VAC.

9. (52) If the rotor of the synchro transmitter shown in Figure 8-7 were displaced  $90^\circ$  from its reference position, \_\_\_\_\_ volts would be induced in the S2 winding.

10. (0) In a simple synchro transmitter-receiver system, voltage differences exist between the \_\_\_\_\_ windings of the transmitter and receiver when the system is out of correspondence.

11. (stator) In a simple synchro transmitter-receiver system, stator current flow will be \_\_\_\_\_ when the system is balanced.  
(minimum/maximum)

12. (minimum) Torque produced in a synchro receiver is proportional to the amount of \_\_\_\_\_ between the transmitter rotor and receiver rotor.

13. (error) In a synchro differential transmitter, the stator acts similar to the \_\_\_\_\_ windings of a variable transformer.  
(primary/secondary)

14. (primary) The \_\_\_\_\_ windings of a differential transmitter are connected to the stator windings of a synchro receiver.

15. (rotor) In a simple TX-TDX-TR synchro system with the TDX at its reference position, the rotor of the TR \_\_\_\_\_ follow the field produced by the TX rotor. (will/will not)

16. (will) To add differential transmitter shaft position to the position of the transmitter shaft, you reverse the \_\_\_\_\_ and \_\_\_\_\_ leads of the differential transmitter.

17. (S1-S3, R1-R3) The rotor of a differential receiver \_\_\_\_\_ free to turn. (is/is not)

18. (is) In a simple TX-TDR-TX synchro system connected for addition, the output of the TDR \_\_\_\_\_ change if only one TX input changes. (will/will not)

(will)

## Control Transformer (CT)

There are many applications where synchros are used as follow up links in automatic control systems. Synchro systems alone do not possess sufficient torque to rotate heavy loads. However, they can control power amplifying devices which, in turn, can move such heavy loads. For these applications, servomotors are used. They are placed in a closed-loop servomechanism employing a special type of synchro called a synchro control transformer to detect the difference, or error, signal between the input and output of the loop. A block diagram of such a control transformer is shown in Figure 8-16.

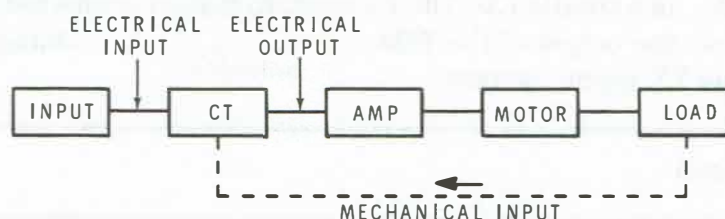


Figure 8-16

Control transformer (CT)  
in a closed-loop system.

### CONTROL TRANSFORMER CHARACTERISTICS

The control transformer is a synchro designed to supply, from its rotor terminals, an AC voltage whose amplitude and phase is dependent on the rotor position, and on the signal applied to the three stator windings. The action of the control transformer in a system differs significantly from that of the other synchro units previously discussed.

Since the control transformer rotor winding is never connected to the AC supply, refer to Figure 8-16, it induces no voltage in the stator windings. As a result, the stator currents are determined only by the voltages applied to them. The rotor itself is wound so that its position has very little influence on the stator currents. Also, there is never any appreciable current flowing in the rotor because its output voltage, referred to as error voltage, is always applied to a high impedance load. Therefore, the rotor does not rotate to any particular position when voltages are applied to the stator windings.

The rotor shaft of the control transformer is always positioned by an external force, and produces varying output voltages from its rotor windings. Unlike either the synchro transmitter or receiver, rotor inductive coupling to S2 is minimum when the control transformer is at its  $0^\circ$  or reference position. In other words, the rotor of the control transformer is displaced  $90^\circ$  from the S2 winding. You can see this by referring back to Figure 8-6.

When current flows in the stator circuit of a control transformer, a resultant magnetic field is produced. This resultant field can be rotated by the output from either a synchro transmitter or a synchro differential transmitter in the same manner as the resultant stator field of the differential transmitter previously discussed. When the field of the control transformer stator is at right angles to the axis of the rotor winding, the voltage induced in the rotor winding is zero — cosine of  $90^\circ$  is 0. When the stator field and the rotor's magnetic axis are aligned, the induced rotor voltage is maximum — cosine of  $0^\circ$  is 1. The control transformer's output is expressed in volts. Therefore, it is convenient to consider its operation in terms of stator voltage as well as in terms of the position of the resultant magnetic field. Remember, however, that it is the angular position with respect to the rotor axis that determines the output.

### SYNCHRO TRANSMITTER — CONTROL TRANSFORMER OPERATION. (TX-CT)

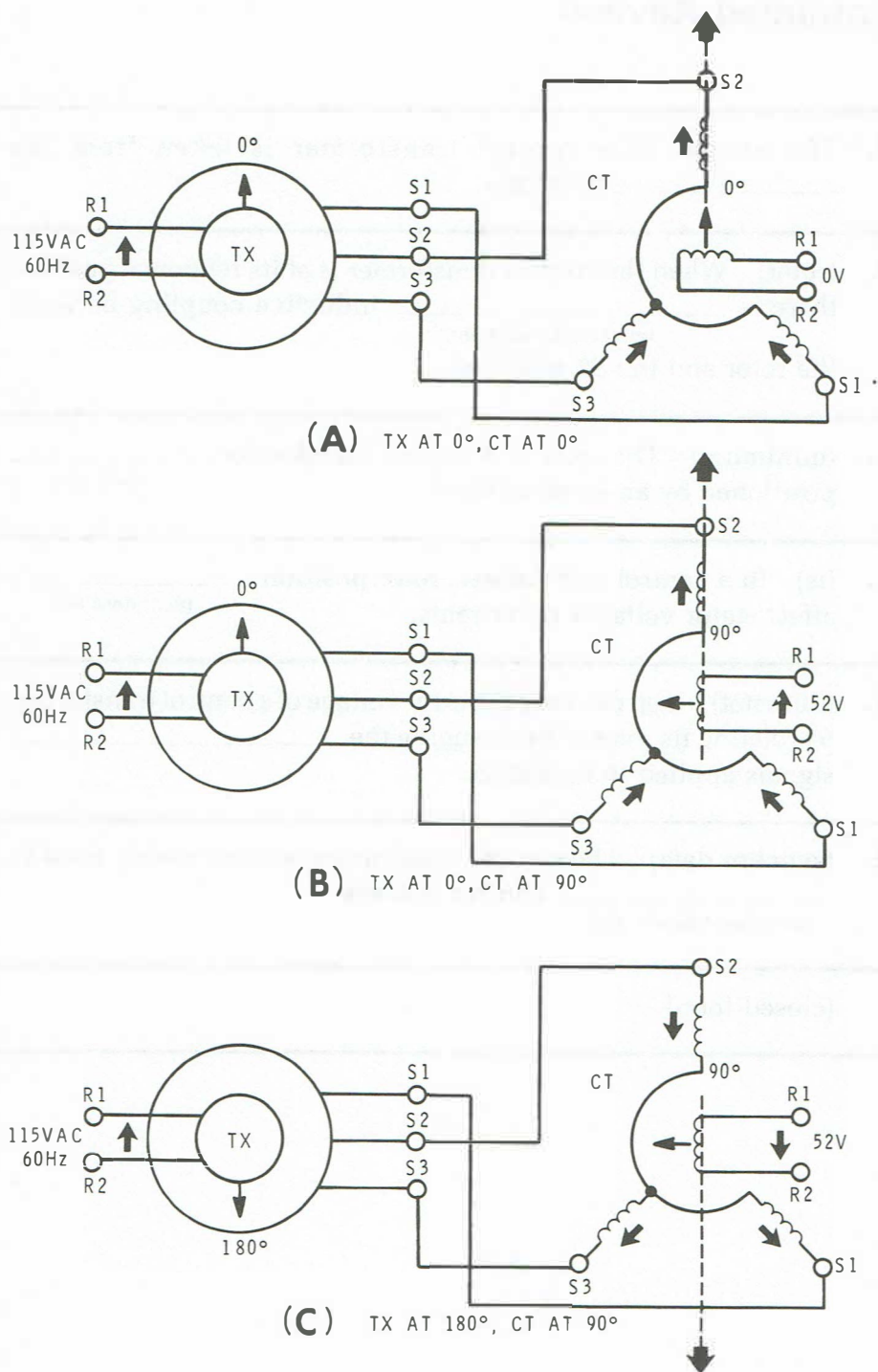
Consider for a moment the conditions existing in the synchro system shown in Figure 8-17A, where the control transformer is connected for operation with a synchro transmitter and the rotors of both units are positioned at their  $0^\circ$  or reference position. The relative phases of the individual stator voltages with respect to the rotor voltage of the transmitter are indicated by the small arrows; while the resultant stator field of the control transformer is shown by the large arrow. With both rotors in the position shown, the control transformer stator field is at right angles to the axis of the rotor coil. Since no voltage is induced in a coil by an alternating magnetic field perpendicular to its axis, the error voltage appearing across the rotor terminals of the control transformer is zero.

Now assume that the control transformer rotor is mechanically rotated  $90^\circ$ , as shown in Figure 8-17B, while the transmitter rotor remains at  $0^\circ$ . Since the control transformer's rotor position does not affect stator voltages or currents, the resultant stator field of the control transformer remains aligned with S2. The axis of the rotor winding is now in alignment with the stator field. Thus, maximum voltage, approximately 52 volts, is induced in the rotor winding and appears across the rotor terminals as the error voltage.

Next, assume the transmitter's rotor is positioned to  $180^\circ$  as seen in Figure 8-17C. The electrical positions of the synchro transmitter and control transformer are  $90^\circ$  apart, the control transformer stator field and rotor axis are aligned, and the control transformer's output is maximum again; but the direction of the control transformer's rotor winding is now reversed with respect to the direction of the stator field. The phase of the output error voltage is therefore opposite to that of the control transformer in Figure 8-17B. This means that the phase of the control transformer's error voltage indicates the direction in which the control transformer rotor is displaced with respect to the synchro data signal applied to its stators.

It is evident that you can vary the control transformer's error voltage by either rotating its rotor or by changing the synchro data signals applied to its stators. You can also see that the amplitude and phase of the error voltage depend on the relationship between signal and rotor, rather than on the actual position of each. You will see in our discussion of servomechanisms the important role the control transformer plays in closed-loop control systems.





**Figure 8-17**  
Synchro transmitter — control transformer operation:  
(A) TX at 0°, CT at 0°;  
(B) TX at 0°, CT at 90°;  
(C) TX at 180°, CT at 90°.

## Programmed Review

19.	The output of a control transformer is taken from the _____ terminals.
20.	(rotor) When the control transformer is at its reference position there is _____ inductive coupling between the rotor and the S2 winding. (minimum/maximum)
21.	(minimum) The rotor of a control transformer _____ positioned by an external force. (is/is not)
22.	(is) In a control transformer, rotor position _____ affect stator voltages or currents. (does/does not)
23.	(does not) You can vary the error voltage of a control transformer by rotating its rotor or by changing the _____ signals applied to its stators.
24.	(synchro data) The control transformer is most widely used in _____ control systems. (open-loop/closed-loop)
(closed-loop)	

## SERVOMECHANISMS

A servomechanism is essentially a high-gain power amplifier operating in an open-loop or a closed-loop configuration. The action of the power amplifier is governed by an error voltage which exists when there is an angular difference between the controlling quantity and the controlled quantity. Because of the high power amplification factor associated with servomechanisms, 10,000:1 is not uncommon; a very small mechanical input can control a very heavy load. Both open-loop and closed-loop servo systems are used for control purposes, but the closed-loop system provides a much greater degree of control. Since open-loop servomechanisms are relatively easy to understand, we will briefly discuss them first and then take a more detailed look at the closed-loop system.

## Basic Servomechanism (Open-Loop) Operation

A basic servomechanism is shown in block diagram form in Figure 8-18A and schematically in Figure 8-18B. As you can see, two synchro units, the synchro transmitter, and the control transformer are used as the input section of the servomechanism.

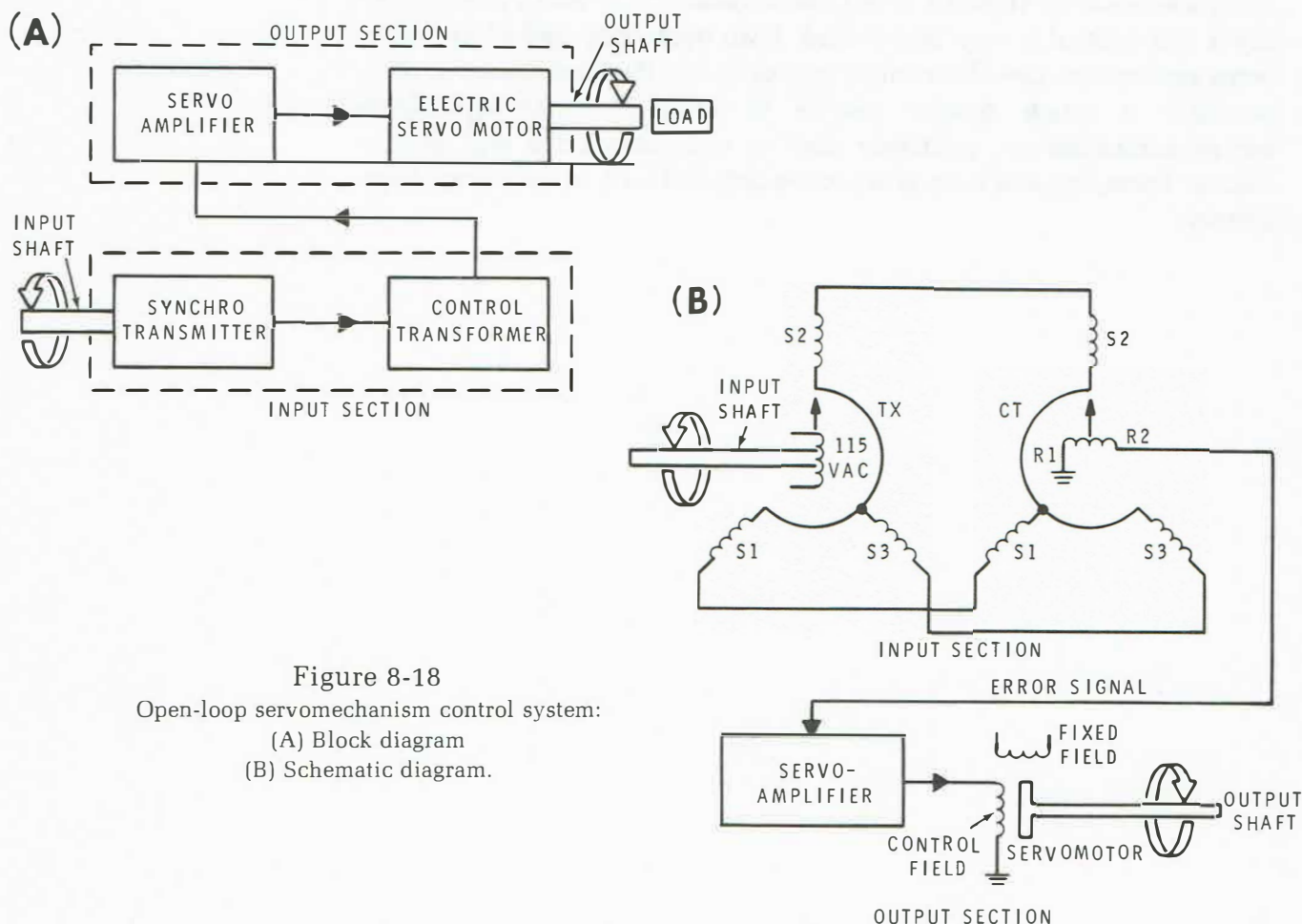


Figure 8-18  
Open-loop servomechanism control system:  
(A) Block diagram  
(B) Schematic diagram.

The input section, which consists of a synchro transmitter and a synchro control transformer, develops the command signal which controls the system's movement. The synchro transmitter's rotor is attached to a shaft, which is rotated by hand or by a controlling mechanical device. If you move the transmitter rotor from its reference position, the voltages in stator windings S1, S2, and S3 will become unbalanced. These voltages, in turn, will be induced in the rotor windings of the control transformer.

The **amplitude** of the induced voltage depends on the **amount** the transmitter's rotor is displaced from its reference; the **phase relationship** of the induced voltage depends on the **direction** the rotor is displaced from its reference. This voltage represents the error voltage.

Since a control transformer is not designed to furnish enough power to drive a load of any significant size, the error voltage must be amplified before it is powerful enough to drive the servo motor and thus move the load. The servo amplifier may be an electronic type, a magnetic type, or a combination of both. In the combination type, the electronic section amplifies the error voltages and the error voltage in turn controls a magnetic field. The power amplification occurs in the magnetic amplifier section. Regardless of the type of amplifier used, the output of the power amplifier is connected to a servo motor. In the single-phase induction motor shown in Figure 8-18, the fixed field is energized at all times but is insufficient to turn the motor's shaft without the aid of the control field. The control field is energized only when an error voltage appears at the control transformer; thus, when the control field is energized, the motor turns.

The direction of the rotation of the motor is determined by the phase relationship of the voltage applied to the fixed field to that of the control field. Varying phase relationships occur only in the control field because its voltage amplitude and phase relationship is determined by the direction of displacement of the synchro transmitter rotor.

Once the input section of the system produces an error voltage, the servo motor continues to turn until the rotor of the synchro transmitter is again positioned to its reference position. This is called an "open-loop" control system, because the servo output has no means of changing the input during operation.

## Closed-Loop Servomechanisms System Operation

Assume for a moment that the load in Figure 8-18 is mechanically coupled back to the control transformer rotor. In that case, as the load rotates in response to transmitter rotation, the CT rotor is also forced to follow this rotation, lagging the transmitter by only a very small angle. When the input rotation stops, the CT rotor, driven by the output servo motor, will quickly drive the small amount necessary to reach its zero position. The rotor of the CT is again perpendicular to the stator field set up by the synchro transmitter and the CT error voltage is zero. Thus, there



is no input to the servo amplifier which, in turn, results in no output to the motor's control field. Therefore, the motor stops rotating. This is an example of a very basic closed-loop system; however, it has little system control.

Figure 8-19 shows a closed-loop servomechanism system that is much more widely used. Note that it is similar to the open-loop system except that it contains an additional follow-up signal stage between the servo output and the amplifier input.

In the closed-loop control system, the servo motor can be stopped at any position without returning the rotor of the transmitter to its reference position. To accomplish this, an error voltage controlled by the servo motor is necessary. This error voltage is called a **follow-up** voltage.

To provide the follow-up voltage, there is usually a bridge element or a control transformer driven by the servo motor's output shaft. The electrical connection is such that the bridge or control transformer generates a voltage opposing the error voltage of the input control section.

The follow-up control transformer is basically the same in construction as those discussed earlier. Note that the stator is excited by the same AC voltage as the rotor of the synchro transmitter. This voltage, however, is only connected across two of the stator windings. The third stator winding is not used. With a voltage applied across the two windings  $120^\circ$  apart, a resultant magnetic field appears midway ( $60^\circ$ ) between the two windings. If the rotor of the follow-up control transformer is placed so that its axis is  $90^\circ$  from the field axis, the voltage induced in the rotor from the two stator windings is zero. However, if the rotor is moved from this zero-voltage position, a voltage will be induced into the rotor. The amplitude of the induced voltage is dependent upon the amount of displacement, and the phase relationship is a function of the direction of rotation of the rotor from its reference position.

When a follow-up control transformer is used in a servomechanism, the rotor is usually driven, through a gear train, by the servo motor's output shaft. The gear ratio is such that the output shaft will rotate many times before the CT rotor is turned any appreciable amount. This greatly enhances the accuracy of the system. The servo motor's output shaft also drives the load.

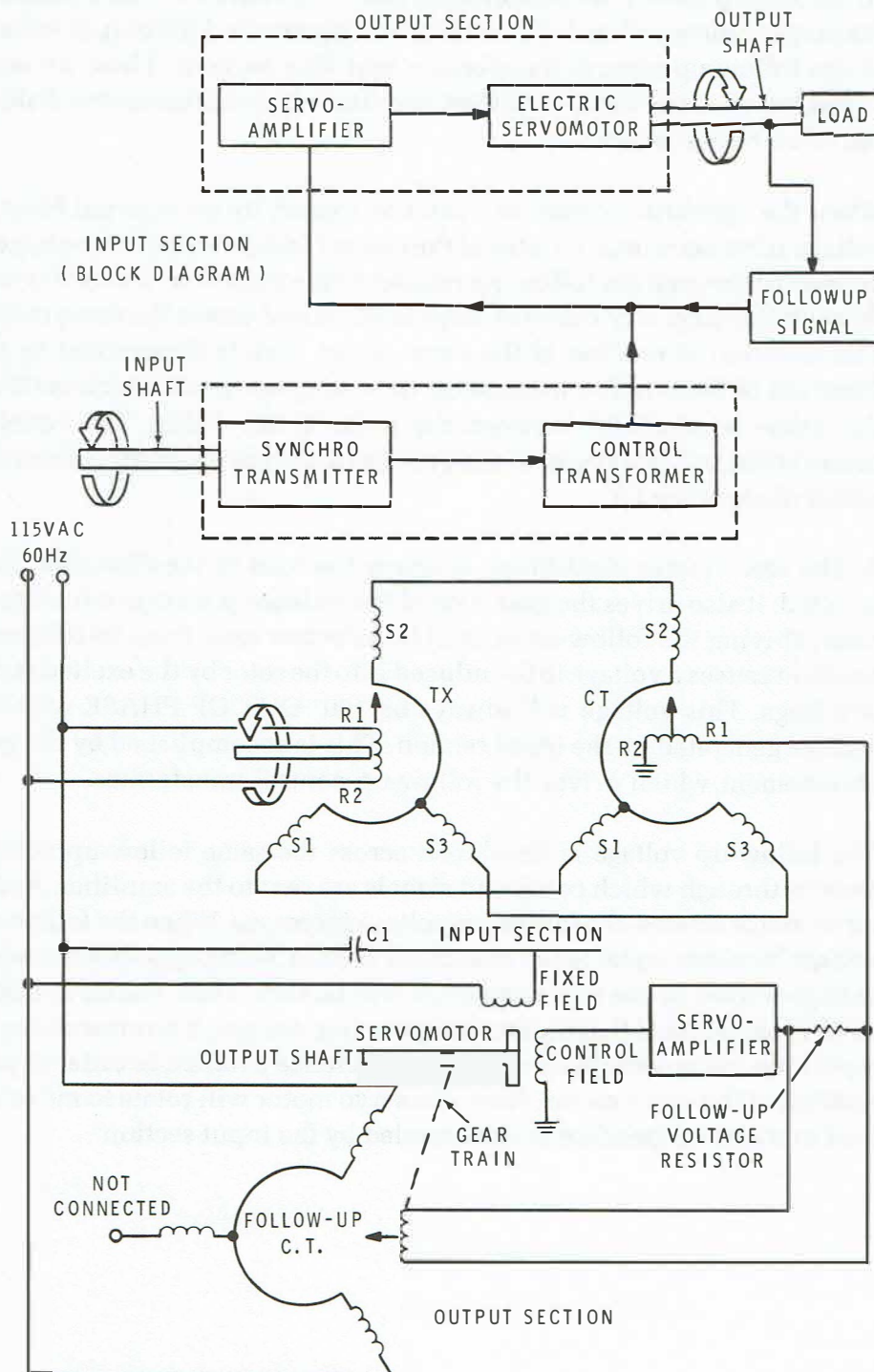


Figure 8-19  
Closed-loop servomechanism control system.

At the static position, with all units at their respective reference position, the output voltage of the input section will be zero and the output voltage of the follow-up control transformer will also be zero. Thus, no error signal is sent to the servo amplifier, and the voltage to the control field of the servo motor is zero.

When the synchro transmitter's rotor is rotated by an external force, a voltage is induced into the rotor of the control transformer. This voltage is connected through the follow-up resistor to the amplifier, where it is sent through the necessary stages of amplification and drives the servo motor. The direction of rotation of the servo motor shaft is determined by the direction of the synchro transmitter rotor displacement, which controls the phase relationship between the motor's two fields. The variable control field will always lead or lag the fixed field by about  $90^\circ$ , due to the action of capacitor C1.

As the servo motor shaft turns, it drives the load in the direction commanded. It also drives the gear train of the follow-up control transformer rotor. Driving the follow-up control transformer rotor from its reference position causes a voltage to be induced into the rotor by the excited stator windings. This voltage will always be  $180^\circ$  OUT OF PHASE with the voltage generated by the input section. This is accomplished by the gear arrangement which drives the follow-up control transformer.

The follow-up voltage is developed across the same follow-up voltage resistor through which command signals are sent to the amplifier. As the servo motor rotates, the follow-up voltage increases. When the follow-up voltage becomes equal to the command voltage, being opposite, the error voltage sensed by the servo amplifier will be zero. Thus, the servo motor stops rotating. Note that unlike the open-loop system, it is unnecessary to reposition the transmitter's rotor to its reference position in order to stop rotation of the servo motor. Here, the servo motor will rotate to move the load to whatever position is commanded by the input section.

## Programmed Review

25. The \_\_\_\_\_ is essentially a high gain power amplifier used to position heavy loads.

26. (servomechanism) The amplitude of the voltage induced in a control transformer is determined by the \_\_\_\_\_ the transmitter's rotor is displaced from its reference.

27. (amount) The phase relationship of the voltage induced in a control transformer is determined by the \_\_\_\_\_ the transmitter's rotor is displaced from its reference.

28. (direction) In a servomechanism, the output of the control transformer is fed to a \_\_\_\_\_ before it is applied to the servo motor.

29. (servo amplifier) The error voltage from the control transformer is ultimately applied to the \_\_\_\_\_ field winding of the servo motor. (fixed/control)

30. (control) In an open-loop control system, the servo motor output \_\_\_\_\_ used to control the input. (is/is not)

31. (is not) In a closed-loop control system, the servo motor can be made to stop at any position without returning the rotor of the \_\_\_\_\_ to its reference position.

32. (transmitter) The error voltage controlled by the servo motor is called the \_\_\_\_\_ voltage.

33. (follow-up) In a closed-loop servomechanism, the follow-up voltage \_\_\_\_\_ the voltage generated in the input section. (aids/opposes)

(opposes)



## DIGITAL-TO-ANALOG CONVERTERS

The world as we know it is an **analog** world. As the sun begins to rise in the morning, the day **gradually** get brighter. As the sun begins to set in the evening, the daylight **gradually** grows dimmer. At the same time, the sun rises, the temperature **gradually** rises then declines towards evening. The sunlight, darkness and temperature are all **gradual** and not abrupt processes. The sunlight doesn't appear at an instant in the morning; then go out instantaneously in the evening. The same is true of starting and stopping an automobile. The car is gradually increased to the speed limit, then the brakes are gradually applied to stop the car. Most people do not floor the gas pedal and slam on the brakes. All of the above are typical examples of **analog** processes. If these processes were represented by a voltage or current, the voltage or current level might gradually increase to some level then gradually decrease. The voltage or current waveform would be continuous and would follow the analog process. Therefore, we would refer to the representative voltage or current waveforms as **analog signals**.

As you are aware, the digital microcomputer does not operate on this principle. The digital world is a **two-state** world made up of 1's and 0's. Therefore, in order for a microcomputer to communicate with the analog world, a digital-to-analog (D/A) or analog-to-digital (A/D) conversion must take place. These two conversions are probably the most important processes related to parallel I/O operations with a microcomputer. Measuring temperature with your microcomputer requires an A/D conversion. However, controlling a thermostat with the same microcomputer requires a D/A conversion. Devices that provide D/A conversions are called **DACs**, and devices that provide A/D conversions are called **ADCs**.

The general operating principles of DACs will be discussed in this section, and ADCs in the next section. In addition, you will study the basic hardware and software requirements for interfacing DACs and ADCs to a microcomputer. Finally, several applications of both DAC and ADC devices will be presented.



## General DAC Concepts

The digital-to-analog converter, or DAC, is normally a monolithic or hybrid device that **converts** the digital output of a computer to a continuous voltage or current signal called an **analog** signal. The analog output is proportional to the digital input value.

DACs are generally classified as being voltage output or current output devices. They are specified by the number of digital input bits and output voltage or current range. Each voltage output value within this range is proportional to a digital input word. For example, suppose an 8-bit DAC has an output voltage range from 0 to 5 volts. In this example, there are 256 ( $2^8$ ) different input words. Therefore, there would be 256 possible output voltage levels including zero. This output would therefore be divided into 255 individual **steps** of  $\frac{5 \text{ V}}{256}$  or 0.0195 volts (19.5 millivolts). Therefore, the DAC has a **step size** of .0195 volts.

If this DAC were driven with an 8-bit free-running binary counter, its output would look like a staircase with 19.5 mV steps as shown in Figure 8-20. Note that the staircase increases to a maximum level representing a binary input of 1111 1111. It then drops to zero when the counter rolls over. Also, note that the maximum output voltage is **not** 5 volts. This is due to the fact that each digital input bit is **weighted** according to its position within the binary input. The least significant bit (LSB) has a weight of  $\frac{5 \text{ V}}{256} = .0195$  volts, the next most significant bit has a weight of  $\frac{5 \text{ V}}{128} = .039$  volts, the next has a weight of  $\frac{5 \text{ V}}{64} = .078$  volts, and so on, with the most significant bit (MSB) having a weight of  $\frac{5 \text{ V}}{2} = 2.5$  volts. If you add all the individual bit weights, you get a 4.98 V maximum output when the DAC input is 1111 1111.

Before we discuss different types of DACs and how they operate, we will define some general terms that are associated with all DACs.

**Resolution:** the smallest analog output change that can occur as result of a change in the digital input.

The resolution, in volts or amperes, is always the **step size**. The step size is always the weight of the least significant bit (LSB) of the DAC input. Thus, the resolution for the DAC in Figure 8-20 is .0195 volts.

Sometimes, DAC resolution is expressed as a percentage of the maximum rated output. To find the % resolution, you divide the step size by the maximum rated output value and multiply by 100. Thus, the % resolution for the DAC in Figure 8-20 would be:

$$\% \text{ resolution} = \frac{\text{step size}}{\text{max. rated output}} \times 100 = \frac{.0195 \text{ V}}{5 \text{ V}} \times 100 = .39\%$$

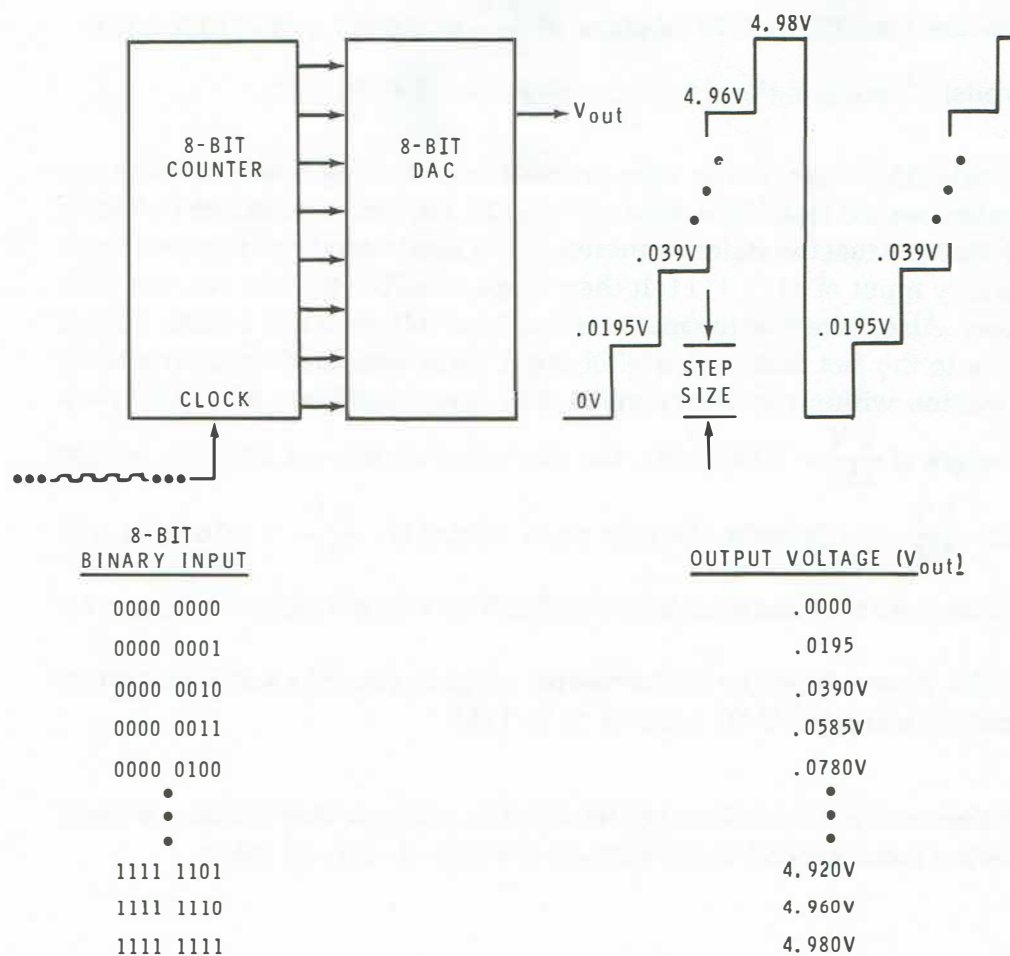


Figure 8-20  
Staircase output from a 0-5 V 8-bit DAC being driven with a free-running 8-bit counter.

An important thing to note here is that, given a particular DAC output range, its resolution increases with the number of input bits. Thus, a 0-5 V 10-bit DAC would have better resolution than a 0-5 volt 8-bit DAC. For this reason, some manufacturers specify resolution as the **number of input bits**.

**Settling time:** the time it takes for a changing DAC output to settle to 99.95 per cent of its new value.

Each digital-to-analog conversion takes time since no process is immediate or instantaneous. The amount of time it takes for a DAC output to stabilize to a new value is called **settling time**. This specification limits how fast conversions can be made and, thus, limits the maximum frequency out of the DAC. Settling times for most standard DACs range from 50 nanoseconds (50 ns) to 20 microseconds (20  $\mu$ s). Typical settling times for current output DACs are around 300 nanoseconds. Voltage output DACs are much slower, typically ten times slower, than equivalent current output devices. Therefore, the majority of monolithic DACs are of the current output variety.

**Accuracy:** the actual analog output obtained from a DAC compared to the expected output.

The accuracy, or **linearity**, of a DAC is usually expressed as a percentage of the maximum rated output value. Therefore, if a 0-5 V DAC has an accuracy of  $\pm 1\%$ , the actual output will be within  $\pm (.19 \text{ times } 5 \text{ V})$  or  $\pm 9.5 \text{ millivolts}$  of the expected value. You might note that given the 0-5 V 8-bit DAC discussed earlier, this is almost 50% of the DAC's step size or resolution.

Of course, the cost of the DAC increases with increased accuracy. When you are choosing any device such as a DAC, you must remember that the **application dictates**. The biggest, fastest, and most accurate are not always the best. It would make little sense to require a DAC with a 50 ns settling time to control a thermostat that might need to be updated only once every 5-10 minutes. By letting the application dictate the design specifications, you can always assure the most cost effective system to perform the given task.

## Types of DACs

Given a digital input to a DAC, there are two common methods of converting this input to an analog voltage or current output. One method is called the **binary weighted** method and the other is referred to as the **R-2R** method. Both use resistor/current divider ladders to translate a binary input bit to a weighted current. The sum of all the ladder branch currents forms the output current.

A binary weighted ladder is shown in Figure 8-21. Note that each resistor in the ladder is twice as big as the one preceding. Therefore, the current produced in each successive current branch is one-half the value of that produced in its preceding branch. The switches are actually solid-state switches (transistors) inside the DAC that are controlled by the binary input word. A binary 1 will produce a branch current with a binary 0 producing no current. The most significant branch current is the largest and, thus, uses the smallest resistor ( $R$ ). The least significant branch current is the smallest and, thus, uses the largest resistor  $2^{n-1}R$ . As you can see, the resistors and currents will be **weighted** by powers of 2.

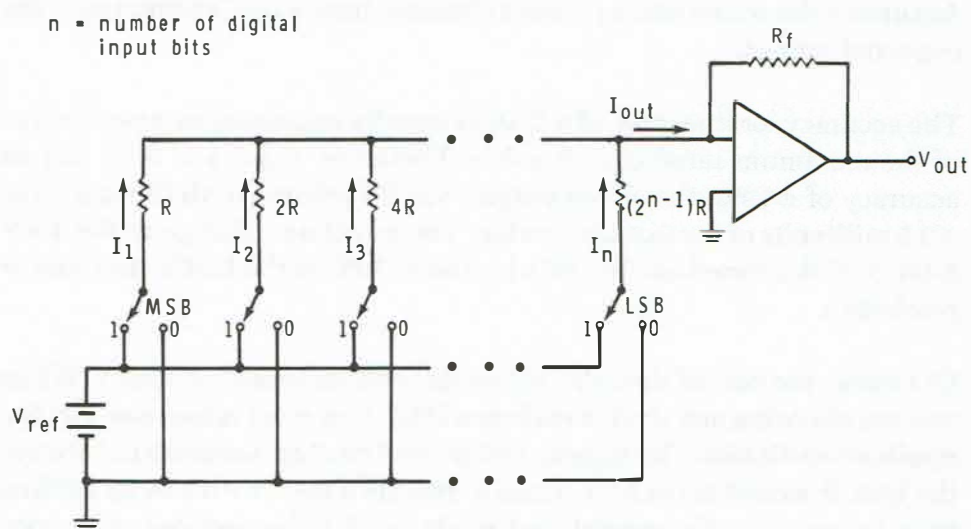


Figure 8-21  
DAC binary weighted ladder circuit.

The sum of all the individual branch currents form the output current ( $I_{OUT}$ ). If the DAC provides a voltage output, a current-to-voltage transducer, such as a parallel-parallel configured op-amp, is added to convert the current output ( $I_{OUT}$ ) to a voltage output ( $V_{OUT}$ ). Notice that the branch currents are produced from a source labeled  $V_{REF}$ . In fact, the voltage output ( $V_{OUT}$ ) is a linear product of the binary input word and the **reference voltage** ( $V_{REF}$ ). More will be said about this later.

You can construct a simple binary weighted DAC with discrete components and interface it to the peripheral interface adapter (PIA), a device that interfaces the MPU with the outside world, as shown in Figure 8-22. This circuit uses port A to supply the digital input value and will provide 6-bit DAC resolution. The PA0 line of port A supplies the least significant digital input bit and PA5 supplies the most significant bit. A CMOS hex buffer such as a MC14050B is used to boost the TTL signal level and supply the reference voltage to the resistor network. In addition, the CMOS buffer can operate above TTL levels and, thus, supply a higher reference voltage to the resistor network. The result is higher current values in the resistor branches that are therefore less subject to noise. Also, the reference level can be adjusted, since the hex buffer can operate anywhere between  $-5\text{V}$  to  $+18\text{V}$  DC. The output current of this circuit will be proportional to the digital input value supplied by port A and, thus, provide the required conversion.

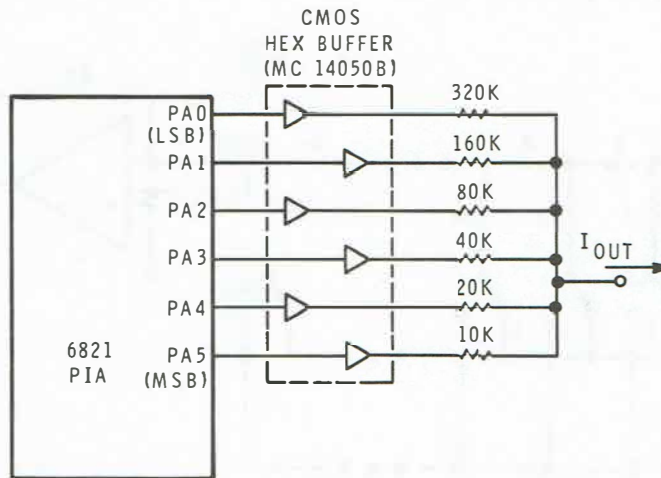


Figure 8-22

A simple 6-bit DAC circuit made of discrete components interfaced to a PIA.

The binary weighted ladder is fine for DACs that have few digital input bits. However, in 8-bit, 10-bit, 12-bit, 14-bit, etc. DACs, the least significant bit branch resistance gets too large and its resulting current too small, and therefore, it is subject to noise. To compensate, you could make the most significant bit branch resistor ( $R$ ) very small. However, this creates the opposite problem — too large of a current in this branch. Another problem is stability. Since there is such a large difference between the least significant resistor value and the most significant resistor value, the transistor switches (buffers) will carry different levels of current, resulting in an unstable output current with temperature and load changes.



The solutions to the inherent problems of the binary weighted DAC are found in the R-2R DAC shown in Figure 8-23. Here, the maximum difference between any two resistors in the ladder is only a factor of 2. The resistors are typically between 100 and 1000 ohms. Therefore, manageable currents are produced. A very common 8-bit DAC, the MC1408, uses 400 ohms and 800 ohms for the values of R and 2R respectively. In addition, since the transistor switches in the DAC chip all carry similar currents, better stability is achieved.

You can see that the R-2R resistor ladder network of Figure 8-23 results in the same binary weighting of currents as in the binary weighted ladder. Therefore, the final output current ( $I_{OUT}$ ) will be the weighted sum of the individual branch currents according to the digital input value to the DAC. Again, a current-to-voltage transducer op-amp can be added to the R-2R ladder network to produce a proportional output voltage. Most commercial DACs use R-2R ladder networks.

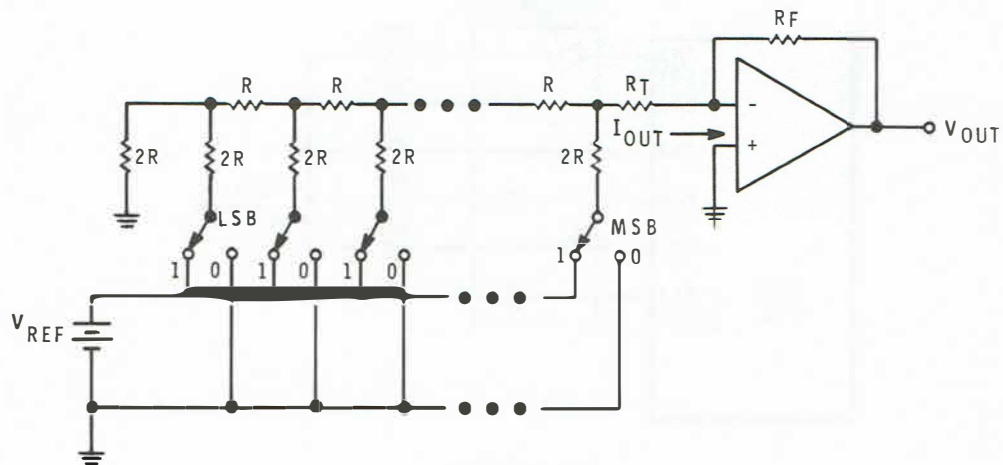


Figure 8-23  
DAC R-2R ladder circuit.

From Figures 8-22 and 8-23, you can see that the output of both types of DACs is a function of the binary input word **and** the reference voltage ( $V_{REF}$ ). In fact, given a reference voltage value and an n-bit DAC, you can calculate its output for a particular binary input by the following equation (assuming unity gain within the DAC):

$$V_{OUT} = V_{REF} \times \sum_{i=1}^n \frac{b_{n-i}}{2^i}$$

where:  $V_{OUT}$  = output voltage

$V_{REF}$  = DAC reference voltage

$n$  = number of bits in the input word

$b_{n-i}$  = value of the **(n-i)th** bit (1 or 0)

For example, suppose you have an 8-bit R-2R DAC with a reference voltage of 5 volts. What would be the DAC output for an input word of 1101 1001? From the above equation, the output would be:

$$\begin{aligned} V_{OUT} &= 5 \text{ V} \left( \frac{1}{2^1} + \frac{1}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} \right) \\ &= 4.238 \text{ volts} \end{aligned}$$

You might verify that the **maximum** output voltage would be 4.98 volts with the DAC in this example. Furthermore, the step amplitude or resolution of the above DAC would be:

$$\frac{V_{REF}}{2^n} = \frac{5 \text{ V}}{2^8} = .0195 \text{ volts}$$

With some DACs, you must supply the reference voltage ( $V_{REF}$ ) externally. Such DACs are called **multiplying** DACs (MDACs). **Non-multiplying** DACs establish an internal reference voltage and no external reference needs to be supplied. In other DACs, a reference voltage is developed internally and brought outside. You can then jumper the available voltage to the reference input to operate in the non-multiplying mode, or use the reference voltage to operate in the multiplying mode. In this way, the DAC can be used in either the multiplying or non-multiplying modes.

Also, by applying an external analog signal to the reference voltage input of a multiplying DAC, you can make a programmable gain amplifier or attenuator. This application will be discussed shortly.

## DAC Interfacing

Interfacing a 4-, 6- or 8-bit DAC to your microcomputer is relatively easy, as shown in Figure 8-24. Here, the DAC input lines can be connected directly to port A or B of the PIA. Note that the binary input to the DAC must be latched to allow enough time for the conversion to take place. With the PIA, this is no problem, since both ports (A and B) are latched when used as output ports. However, without the PIA, an external latch is required between the microprocessor and the DAC, although some DACs are **internally buffered**, meaning that the latching capability is internal to the device.

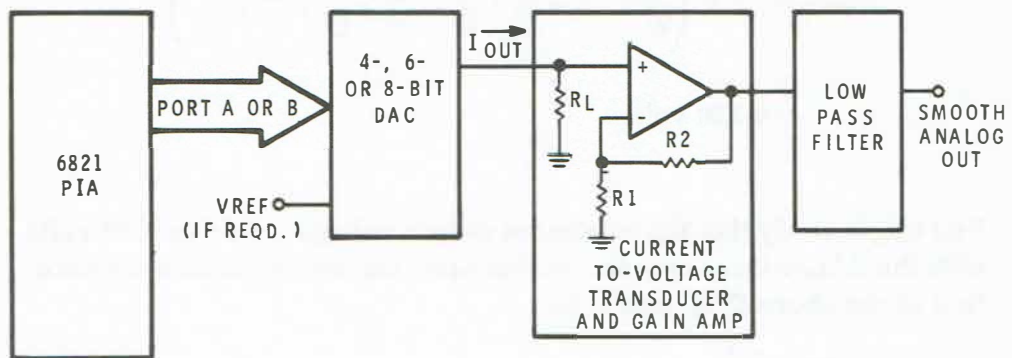


Figure 8-24  
4-, 6- or 8-bit DAC interface to a PIA.

Since most DACs are of low-level current output variety, a current-to-voltage transducer and amplifier will be required to achieve a usable voltage output. A standard 741C op-amp can be used for this purpose. The op-amp circuit shown in Figure 8-24 will provide a voltage output of:

$$V_{OUT} = (I_{OUT} R_L) \left( \frac{R_2}{R_1} + 1 \right)$$

Finally, if a smooth analog output signal is desired, a low-pass filter is required to smooth out the "steps" of the DAC output. The filter cutoff frequency will depend on the frequency of the output signal. In general, the filter cutoff frequency will be equal to the highest frequency component of the generated waveform.

Many applications require a greater resolution than can be provided by a 8-bit DAC. Fortunately, for these applications, there are 10-, 12- and 14-bit DACs commercially available. These DACs may also be easily interfaced to your microcomputer. However, there is one small problem. When interfacing a DAC of more than eight bits to the PIA, you must use both port A and port B. When data is transferred to the PIA ports, both ports will not receive the data simultaneously. For example, suppose that you have interfaced a 10-bit DAC to the PIA such that the port A lines provide the eight least significant DAC input lines and port B lines 0 and 1 (PB0 and PB1) provide the two most significant DAC input lines. Now, suppose the DAC is presently providing an analog signal which corresponds to a 10-bit digital value of 00 1000 0000. It is now desired to convert a new 10-bit value, say 10 0000 0000, which is clearly higher than the present value. If you were to load this new value in the microprocessor index register, then store it to the PIA, what would happen? Since port A forms the least significant 8-bits of the 10-bit value, it would receive its eight new bits first, then the two most significant bits would be stored to port B. However, between the port A and port B operations, an intermediate and undesirable 10-bit data value would appear at the DAC input. This intermediate value would consist of the eight least significant bits of the new value from port A and the two most significant bits of the old value from port B. With the two values we have chosen, the intermediate value would be, 00 0000 0000. Obviously, this is undesirable. The only solution to the problem is **double buffering**. That is, you must **latch** the **entire** 10-bit word before it is seen at the DAC input.

The circuit in Figure 8-25 shows one way that double buffering can be accomplished with the PIA. The 10-bit value can be stored to ports A and B from the 6800 index register. Port A will receive the eight least significant bits first, then port B will receive the two most significant bits. The port B control register will be configured for the partial hand-shake mode such that CB2 will be active low for one MPU cycle when the write-to-port B operation takes place. Thus, when the two most significant bits of the 10-bit value are written to port B, the latches will be enabled and pass the entire 10-bit value to the DAC input lines simultaneously. An inverter has been used in this circuit to convert the CB2 active low pulse to an active high pulse. The active high pulse is required by both the 74100 and 7475 latches.

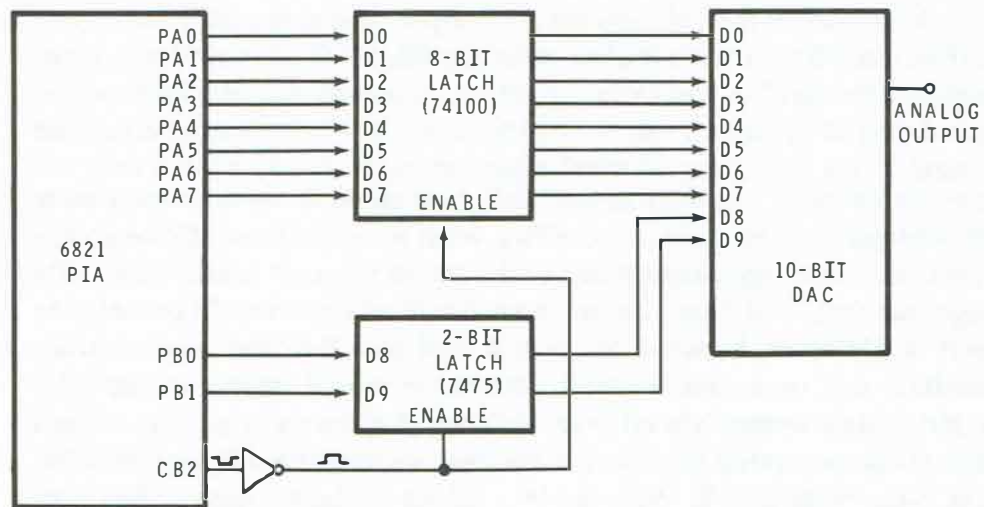


Figure 8-25  
Double buffering for a 10-bit DAC  
using a PIA.

Some manufacturers advertise that their DACs are “microprocessor compatible.” When this is the case, the DACs are usually buffered internally. For less than 8-bits, a single internal buffer (latch) is provided. For more than 8-bits, a double buffer (latch) is provided internally. With these devices, no external latching is required.



## DAC Applications

As mentioned earlier, there are numerous microcomputer applications which require the use of a DAC. Let's take a look at just a few of the more common DAC applications.

### WAVEFORM GENERATION

A simple **ramp** generator is shown in Figure 8-26. Here, the PIA output port is cleared then successively incremented to provide a continuous count to the DAC. The DAC output will cycle through its entire range and increase towards a maximum with each count, then drop to zero when the PIA count returns to zero or rolls over. An inverted ramp can be obtained by using a decrementing routine. However, the incrementing or decrementing routine cannot be faster than it takes for the DAC to make a conversion. Therefore, a short software time delay routine is normally required between the increment or decrement steps in the program.

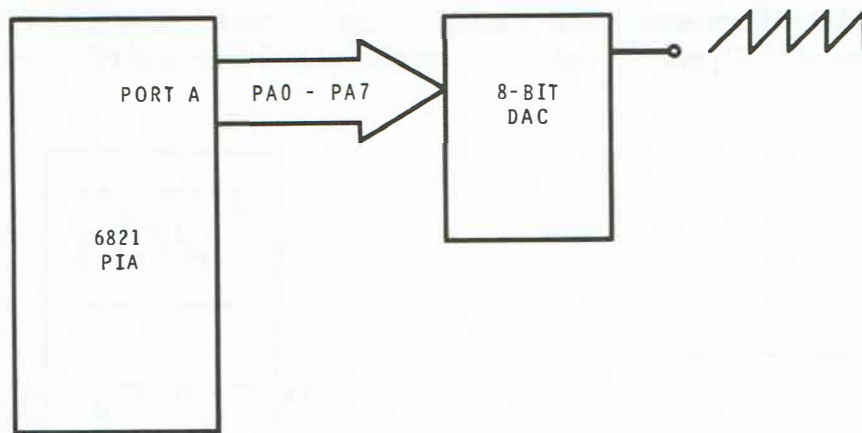


Figure 8-26  
Ramp generator.

As mentioned earlier, the maximum output frequency is limited by the DAC settling time. Most common DACs provide output frequencies up to 30-40 kHz. To get lower output frequencies, you simply lengthen the time delay between increment/decrement operations.

Recall that the DAC output is actually a staircase made up of individual steps. In most applications, this staircase is left unfiltered. However, you can filter the DAC output if the staircase output is unacceptable.

A ramp output is commonly used to drive the horizontal (X) trace on any X-Y display device such as an oscilloscope or X-Y chart recorder.

A triangular wave output can be obtained by incrementing the PIA output register to its maximum value ( $FF_{16}$ ) then decrementing it back to zero and repeating this process.

Square wave outputs can be generated by alternating the PIA output register between zero and some maximum value to obtain a desired output level. The amount of time the output register remains at zero and its maximum value will determine the output frequency and duty cycle.

Finally, complex waveforms may be generated using more sophisticated software routines. In many cases, the software consists of a series of subroutines. The subroutines generate various time delays along with the standard ramp, triangular, and square wave outputs. Complex waveform outputs can then be accomplished by linking these various subroutines together.

Two DACs can be used to display a waveform on an X-Y display device such as an X-Y chart recorder or oscilloscope as shown in Figure 8-27. One DAC will generate a binary ramp for the horizontal (X) input while the second DAC provides the waveform output to the vertical (Y) input.

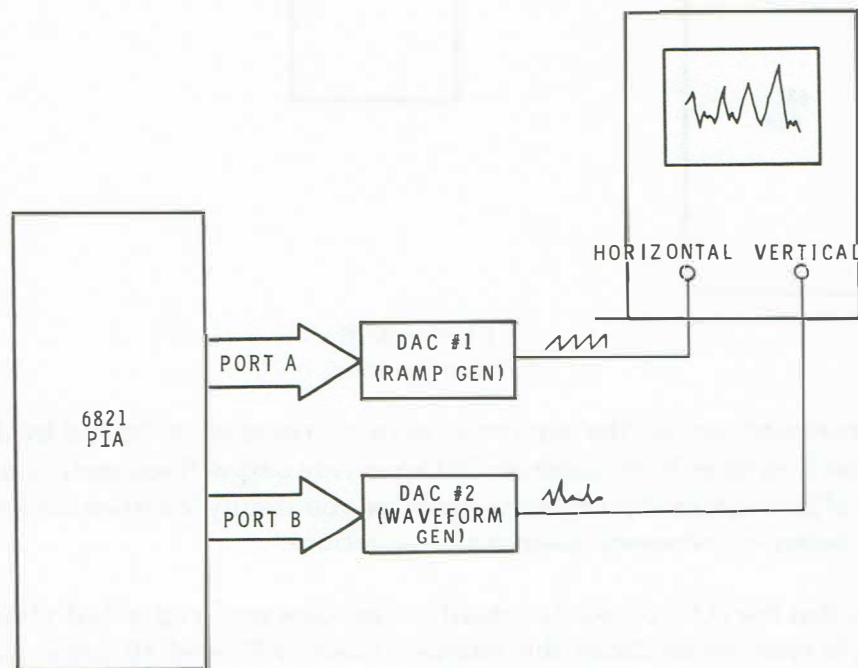


Figure 8-27  
Two DACs being used to drive an  
X-Y device (oscilloscope).

## PROGRAMMABLE GAIN AMPLIFIER AND ATTENUATOR

A programmable gain amplifier or attenuator can be made with a multiplying DAC as shown in Figure 8-28. Recall that, with a multiplying DAC, the analog output is proportional to the product of the binary input and the reference voltage ( $V_{REF}$ ). Thus, by applying a signal to the reference voltage input, the signal can be amplified or attenuated. The DAC output will change along with the input reference signal. By controlling the digital input value to the DAC with the MPU, you can control the amplitude of the output signal and, thus, amplify or attenuate the input reference signal. Since most DACs are current output, a current-to-voltage converter op-amp is normally required to obtain a voltage output.

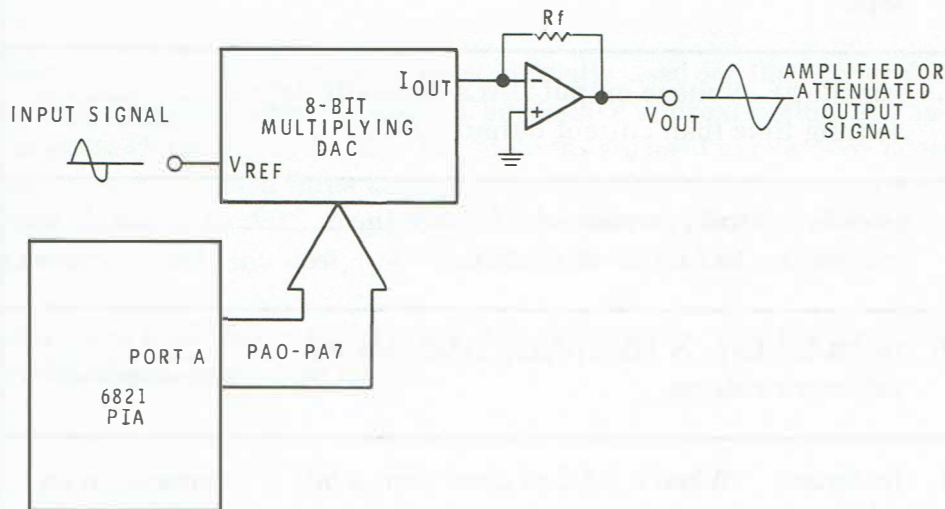


Figure 8-28  
Programmable gain amplifier or attenuator.

## MOTOR CONTROL AND POSITIONING

One of the most common uses of the DAC is for motor control and positioning. The DAC is required when the control or positioning operation is being performed by a digital circuit such as a microcomputer. This application will be discussed in detail in a later unit.

### ADCs

Finally, DACs are used internally as part of **ramp** and **successive approximation** analog-to-digital converters (ADCs). This application is discussed in the next section of this Unit.

## Programmed Review

34.	Continuous voltage or current signals are referred to as _____ signals.
35.	(analog) Digital-to-analog converters are generally classified as being _____ or _____ output devices.
36.	(voltage, current) Resolution is defined as the _____ (smallest/largest) analog output that can occur as a result of a change in the digital input.
37.	(smallest) Voltage output DACs generally have a _____ settling time than current output DACs. (slower/faster)
38.	(slower) Most commercial DACs use the _____ conversion technique internally.
39.	(R-2R ladder) A multiplying DAC uses an _____ reference voltage. (internal/external)
40.	(external) When a DAC of more than 8 bits is interfaced to an 8-bit microcomputer, _____ must be provided between the microcomputer and the DAC input lines.
41.	(double buffering) If you want a smooth analog output signal, you will need a _____ filter to smooth out the steps of (low-pass/high-pass) the DAC output.
42.	(low-pass) DACs are used internally as part of ramp generation and _____ analog-to-digital converters.
	(successive approximation)

## ANALOG-TO-DIGITAL CONVERTERS

Almost all “real world” data, except for financial data, is analog in nature. Therefore, the use of ADCs is quite common in microcomputer systems, especially those systems that are dedicated to monitoring “real world” events. An analog-to-digital converter (ADC) can be a board level, hybrid or monolithic device that converts a continuous voltage signal, or analog signal, into a multi-bit digital word. Data flow through the ADC is just the opposite of that through a DAC.

### General ADC Concepts and Conversion Techniques

There are three common techniques normally used for the conversion process. They are: **ramp generation**, **successive approximation**, and **integration**. Of the three, the later two methods are used in over 95% of all ADCs. There are also three key specifications that must be considered when selecting an ADC for your microcomputer application. They are: **accuracy**, **speed**, and **cost**, not necessarily in that order.

Let's take a brief look at each type of ADC, keeping in mind the three key specifications mentioned above.



## RAMP GENERATION

The analog-to-digital conversion can be accomplished using a DAC and comparator interfaced to the PIA as shown in Figure 8-29. The conversion process is started by clearing port A. The port is then incremented by software to provide a continuous count to the DAC input. This will provide a steadily increasing voltage **ramp** at the DAC output. This ramp voltage is then constantly compared to the unknown voltage until they are equal. When equal, the comparator will provide an active pulse to the CA1 input control line which, in turn, generates an interrupt to the microprocessor to stop the ramping process. A timing diagram of the voltage ramp and compare operation is also shown in Figure 8-29. The final count provided by port A will be the digital equivalent of the unknown analog voltage. This value can then be obtained by reading port A. Recall that, although port A must be configured for output to supply the ramp, you can still obtain the final count (digital conversion value) by reading the contents of the port A output register when the CA1 interrupt occurs.

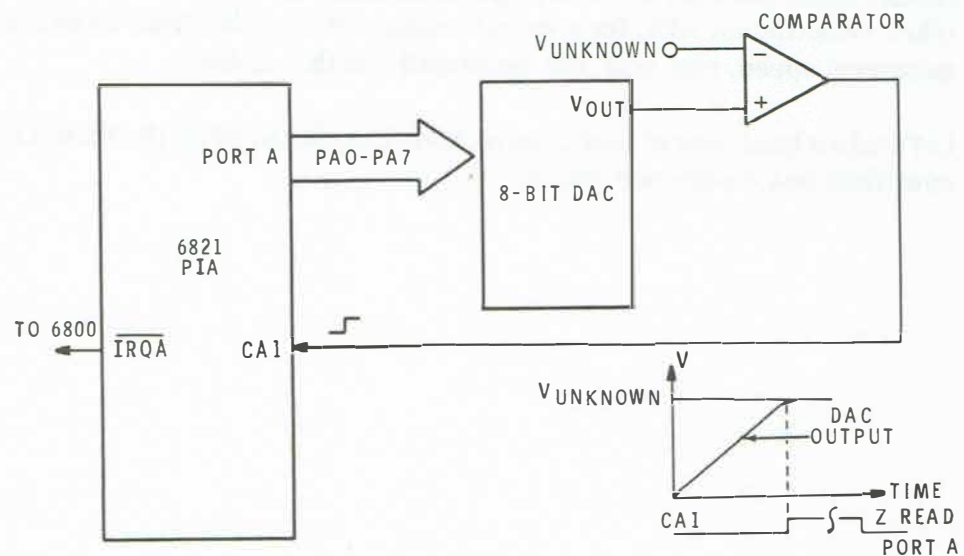


Figure 8-29

A ramp generator ADC using a DAC and comparator interfaced to the PIA.

This method is relatively straight forward and simple; however, it has several disadvantages. First, most DACs are current output devices. Therefore, you must supply a stage which will convert the current output of the DAC to a voltage output whose range will cover all possible unknown input voltage values. Second, you must supply the comparator stage. Both the converter and comparator stages can be simple op-amp circuits. However, the DAC, op-amps, and associated circuitry (resistors, capacitors, etc.) increase the total parts count and, obviously, the cost required to perform the conversion. Third, this method is relatively slow, especially for higher unknown input values within the input range. This is because the DAC output is incremented until the unknown value is reached. In addition, valuable MPU time is being used to perform the conversion. Recall that the MPU must delay, or waste time, between increments since it is much faster than the DAC. In many cases, the MPU is dedicated to the conversion process for many milliseconds. Finally, the conversion could “miss” instantaneous changes. For example, the peak amplitude of an input waveform could be missed as shown in Figure 8-30. Suppose the peak value were detected and used to start the conversion process (via an interrupt). By the time the ramp reached the unknown input voltage level, it would be somewhat different than the desired **peak** voltage level reading.

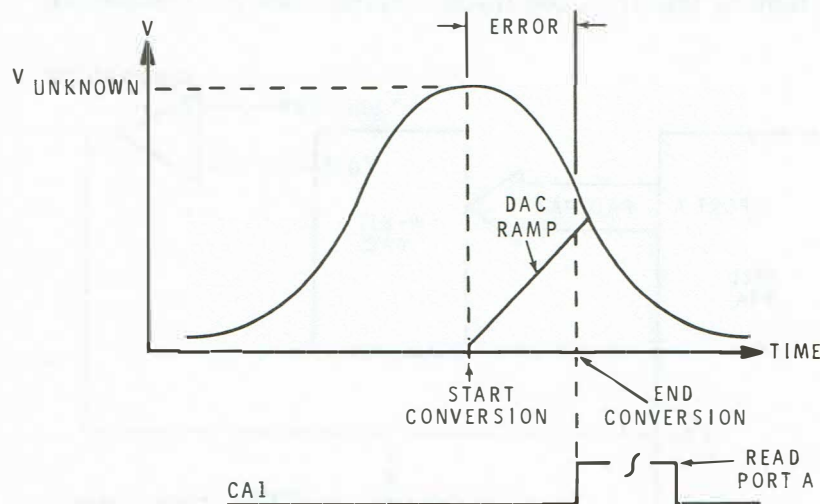


Figure 8-30

Using a ramp ADC to detect peak levels.

For the above reasons, the ramp generation method of analog-to-digital conversion is not widely used in microcomputer systems. As stated earlier, over 95% of analog-to-digital converters use the successive approximation or integration methods.

## SUCCESSIVE APPROXIMATION

The successive approximation method of analog-to-digital conversion is one of the fastest, most accurate, and most widely used A/D conversion techniques. Again, at the heart of the converter is a DAC and a comparator. A successive approximation ADC circuit is shown interfaced to the PIA in Figure 8-31. This is similar to the ramp generator circuit previously discussed. However, rather than providing an increasing ramp output until the unknown level is reached, the DAC “homes in” on the unknown voltage level using a trial and error method. The comparator provides “greater than” and “less than” indications of the DAC output versus the unknown voltage level. A bit-by-bit comparison is made, starting with the most significant DAC input bit and ending with the least significant DAC input bit.

At the start of the conversion, the most significant input bit (MSB) to the DAC (PA7, Figure 8-31) is set, producing a DAC output equal to one-half of its maximum value. This output is compared to the unknown analog value by the comparator. If the DAC output is greater than the unknown value, the PA7 line is cleared. However, if the DAC output is less than the unknown value, the PA7 line is left set. With the circuit in Figure 8-31, PB0 must be configured as an input line and polled to determine the greater than or less than condition coming from the comparator.

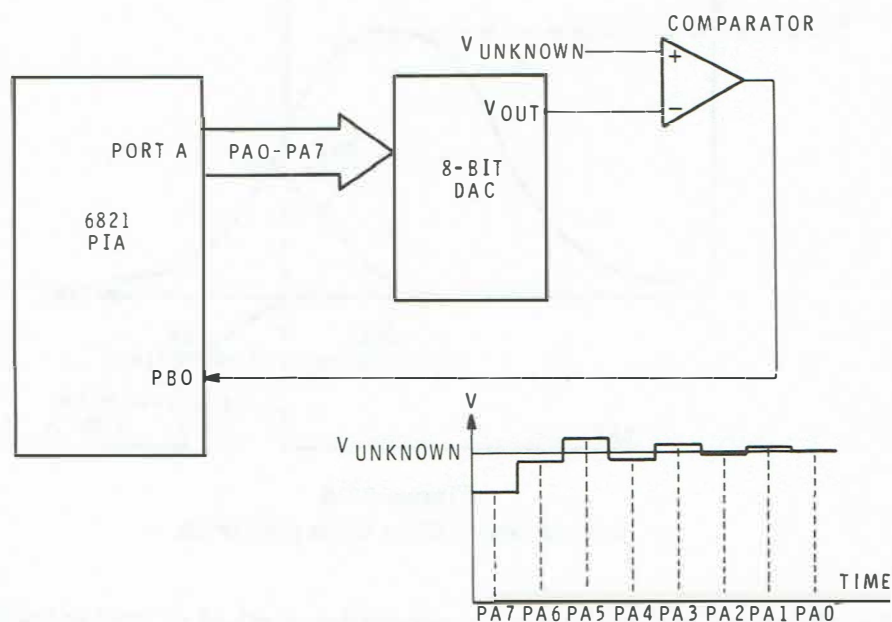


Figure 8-31

A successive approximation ADC using a DAC and comparator interfaced to the PIA.

After the MSB is tested, the next most significant bit (PA6) is set and compared in the same way. The process is repeated until all the DAC input bits have been tested. Once all bits have been tested, a read-port A operation will provide the converted value.

A flow chart for the required MPU software is shown in Figure 8-32. With the successive approximation method, only  $n$ -tests must be made for an  $n$ -bit converter. Therefore, this method is much faster than the ramp method. However, as you can see from Figure 8-32, the required software can get a little tricky. In addition, the MPU is being tied up to perform the conversion.

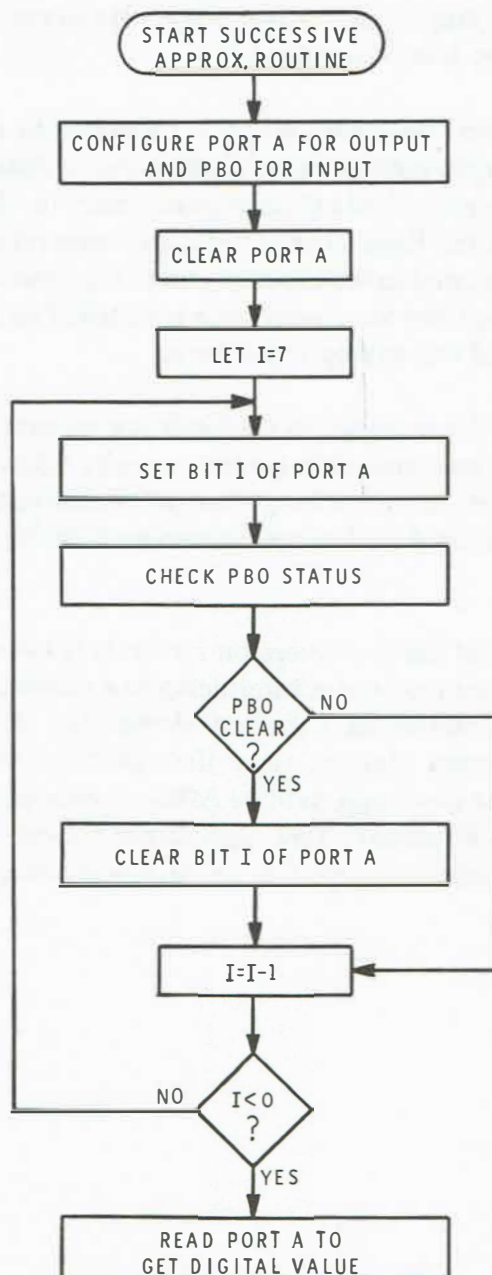


Figure 8-32  
Successive approximation software flowchart  
for the circuit in Figure 8-31.

The successive approximation method is used in many single-chip ADC devices. These devices free the MPU from the conversion task, provide faster, more accurate conversions, and are relatively inexpensive. You will learn how to interface these devices to your microcomputer system shortly.

### INTEGRATION

Another method which is employed by single-chip ADC devices is the **integration** method. There are various forms of this method; namely, single slope, dual slope, and triple slope. However, the **dual-slope** method is by far the most common.

Basically, dual-slope converters allow a capacitor to charge from the unknown analog input voltage for a fixed period of time. The charge on the capacitor at the end of this time depends only on the input voltage level. At the end of the fixed charge time, a known reference voltage of opposite polarity is gated to the capacitor such that it will discharge back to zero. The time required for discharge is counted. The final count is the digital equivalent of the analog input level.

The advantages of the integration methods are accuracy and noise immunity. However, the conversion process usually takes between 10 and 50 milliseconds. This is quite a long time in the computer world. One of the most common uses of dual-slope converters is in digital panel meters (DPMs).

Further discussion of these conversion methods is beyond the scope of this course since microcomputer interfacing to a given application is our primary concern. Interfacing different single-chip ADC devices to a microcomputer is very similar, regardless of the conversion method used. Therefore, our next topic will be ADC/microcomputer interfacing. Consult the Heath **Electronic Test Equipment** course (EE-3105) if you want a more detailed discussion of analog-to-digital conversion techniques.



## Interfacing to ADC Devices

Fortunately, single-chip ADCs are becoming increasingly available at low cost. The devices normally use the successive approximation or dual-slope conversion methods just discussed. The application will dictate the priority of cost, speed, and accuracy and, thus, determine which type of ADC is to be used.

Interfacing these devices to your microcomputer system is relatively easy. An 8-bit ADC is shown interfaced to the PIA in Figure 8-33. Most **microprocessor compatible** ADCs have two major control lines. One line is used to start the conversion process by the microprocessor and the other is used to signal the microprocessor when the conversion is finished. The most common designation for the start line is simply START. However, some manufacturers might designate it as Initiate Conversion (IC), RUN, CONVERT, etc. Various designations such as Data Valid (DV), DONE, STATUS, BUSY, etc. are also used by different manufacturers for the end of conversion (EOC) line.

In Figure 8-33, CB2 of the PIA is used to start the conversion. When the conversion is complete, the ADC will activate the CB1 input line of the PIA to generate an interrupt to the MPU. A read operation can then be performed on port B to input the digital data.

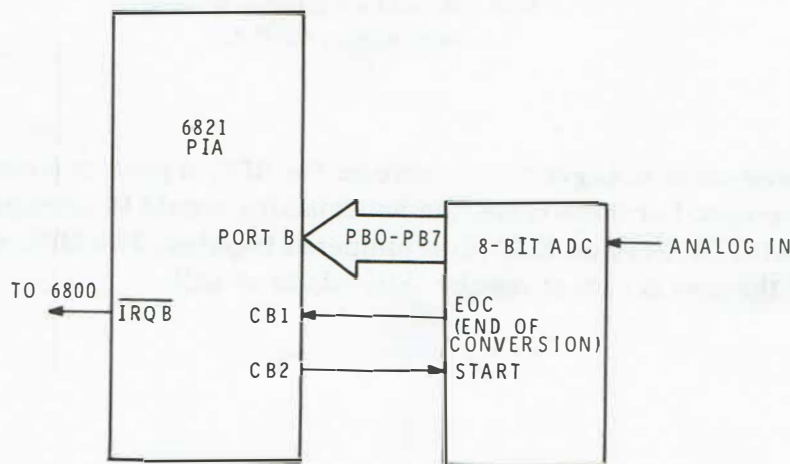


Figure 8-33  
8-bit ADC interfaced to the PIA for  
control of the conversion process.

Most common microprocessor-compatible ADCs have internal output latches. However, if the ADC output is not latched, you will need to provide a latch between the ADC and PIA as shown in Figure 8-34. The latch is required since, without it, the unlatched ADC output data would be lost before the MPU could service the end of conversion interrupt and read the port B data. For input operations, the PIA port data is **unlatched** and external input latching must be provided. Note that the end of the conversion signal in Figure 8-34 will enable the latch and CB1 interrupt input simultaneously.

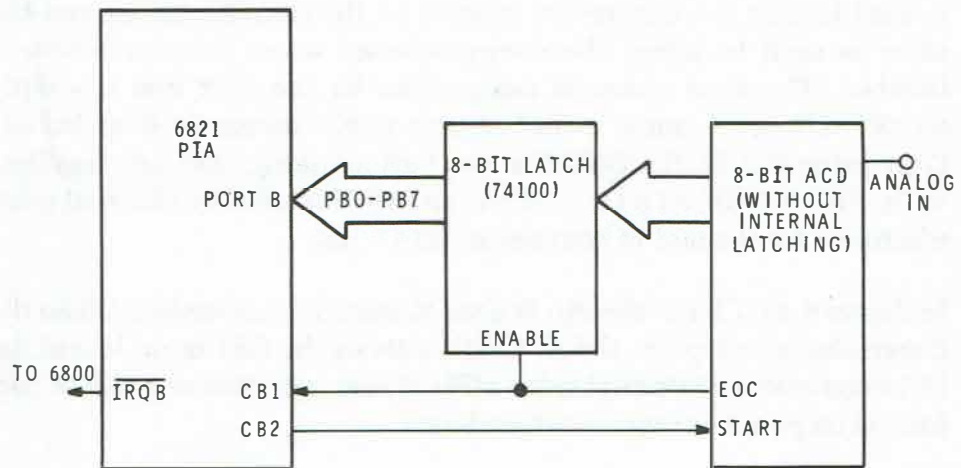


Figure 8-34  
8-bit ADC without internal latching  
interfaced to the PIA.

In some cases, it might be desirable for the ADC to provide a continuous conversion. For these cases, the handshaking would be eliminated and the ADC START and EOC lines jumpered together. The MPU will then read the port B data at regular intervals or at will.

If higher digital resolution, and in turn higher accuracy, of the input analog signal is desired, 10-bit, 12-bit, or 14-bit ADCs can be used. A 10-bit latched ADC is shown interfaced to the PIA in Figure 8-35. Note that the only difference is that port A is being used to input the two most significant bits. Therefore, the digital value will be represented by two bytes of data. If ports A and B are assigned to consecutive memory locations, a load index register (LDX) instruction can be used to input the data to the microprocessor index register. The low byte of the index register ( $X_L$ ) would contain the eight least significant bits (port B data) and bits 8 and 9 of the index register would contain the two most significant bits from port A. The remaining index register bit locations (10, 11, 12, 13, 14, 15) would be ignored.

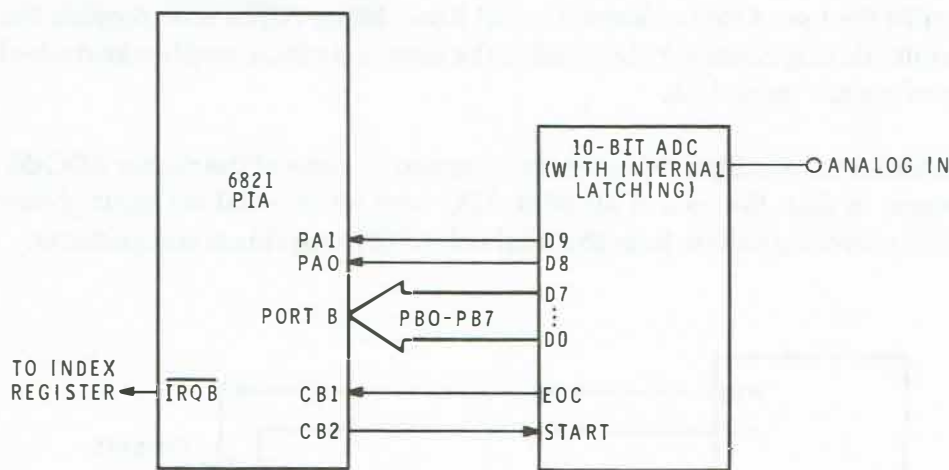


Figure 8-35  
10-bit ADC with internal latching  
interfaced to the PIA.

Finally, many ADCs are used as part of analog data acquisition systems. A data acquisition system (DAS) can be a simple digital panel meter (DPM) which uses a single ADC, or it can consist of an entire multiprocessor system using many ADCs. In many cases, several ADCs are **multiplexed** into a local processor. The local processor collects, monitors (and can provide real-time analysis) of the data. If a production process is being monitored, the local processor might perform various control functions as a result of the data analysis. In this way, a production process can be kept in control. A local processor might also periodically transmit its data to a central processor for further data analysis and long-term storage. In fact, many local processor systems could feed into a central processor and, thus, create a multiprocessor network. Each local processor system is a unique microcomputer system by itself.

Now, suppose that a local processor is required to collect and analyze data from four separate analog sources. This can be accomplished in two ways: by using four separate ADCs or by using one ADC and an analog multiplexer. The later method is the most economical and is usually the method which is employed.

The circuit in Figure 8-36 illustrates the multiplexing method. An 8-bit ADC with internal latching is interfaced to the PIA as before. One of four analog inputs is passed to the ADC via a 4:1 multiplexer. Channel selection is accomplished by using PA0 and PA1 as output lines. You select any one of the four analog inputs by writing the proper 1's and 0's to the PA0 and PA1 output register bits. If more analog signals must be monitored, larger multiplexers are required. Larger multiplexers would require the use of more channel select lines. Many ADCs now provide the multiplexing capability internal to the device, as indicated by the dashed outline in Figure 8-36.

Internal 8:1 multiplexers are very common in some of the newer ADC devices. In fact, the cost of an 8-bit ADC with an internal 8:1 multiplexer will normally be less than the total cost of the individual components.

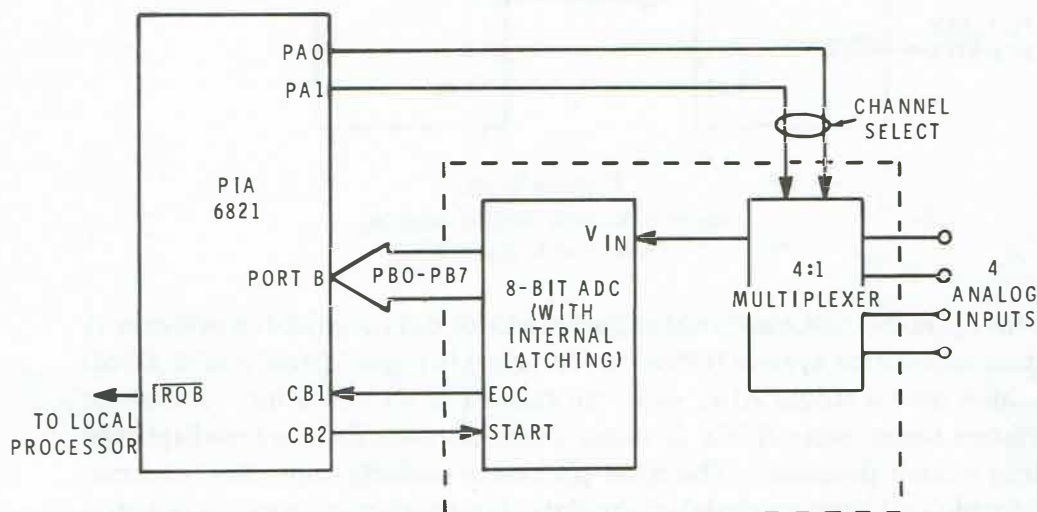


Figure 8-36  
Multiplexed input for an 8-bit  
ADC interface.

## Programmed Review

43.	Ramp generation, successive approximation, and integration are the most commonly used techniques in the _____ conversion process.
44.	(analog-to-digital) The ramp generation and successive approximation conversion techniques use a _____ and a _____ as part of the analog-to-digital conversion.
45.	(DAC, comparator) As compared to ramp generation, successive approximation is _____ and _____ accurate. (slower/faster) (more/less)
46.	(faster, more) A 10-bit successive approximation ADC has to make a maximum of _____ tests on an unknown analog signal to determine its value.
47.	(ten) A digital panel meter usually uses a _____ slope integration type ADC. (single/dual)
48.	(dual) A single-chip ADC is usually _____ expensive and requires _____ software than a discrete design. (more/less) (more/less)
49.	(less, less) Most microprocessor compatible ADCs have two control lines called _____ and End of Conversion (EOC).
50.	(Start) A PIA _____ provide an input latching function. (does/does not)
51.	(does not) When used in conjunction with a multiplexer, an ADC _____ monitor multiple analog inputs. (can/cannot)
	(can)



## EXPERIMENT

Perform Experiment 14. This experiment can be found in Unit 12. After you finish the experiment, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct answers that appear after the exam.

1. Regarding synchro system operation, which of the following is **not** a true statement?
  - A. Synchro systems are usually used as components of a servomechanism.
  - B. Synchro systems are capable of transmitting angular shaft position over long distances.
  - C. Synchro systems provide a high torque output.
  - D. A synchro transmitter acts in a manner similar to a variable transformer.
2. Which of the following synchro devices is used when it is desirable to obtain a voltage output only?
  - A. Differential synchro transmitter.
  - B. Control transformer.
  - C. Synchro receiver.
  - D. Differential synchro receiver.
3. In a synchro system, which of the following synchro devices is used as a system error correction unit?
  - A. Differential synchro transmitter.
  - B. Synchro receiver.
  - C. Differential synchro receiver.
  - D. Synchro transmitter.
4. With the rotor of the synchro transmitter, shown in Figure 8-37, turned  $75^\circ$  in the clockwise direction from its reference position, which of the following voltages will be induced in the S1, S2, and S3 windings respectively?
  - A. 29.8 V, 81.3 V, 111.1 V.
  - B. 36.8 V, 13.5 V, 50.2 V.
  - C. 81.3 V, 111.1 V, 29.8 V.
  - D. 13.5 V, 50.2 V, 36.8 V.

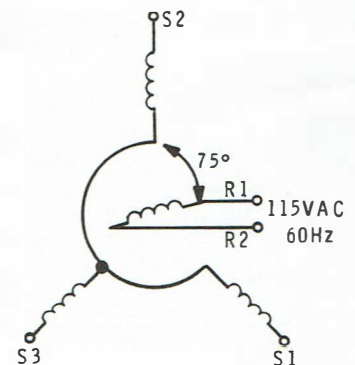


Figure 8-37  
Synchro transmitter.

5. In a simple transmitter-receiver synchro system that is in an unbalanced condition, the voltages induced in the receiver stator windings are:
- A. Equal to the voltages induced in the transmitter stator windings, thus causing current to flow.
  - B. Equal to the voltages induced in the transmitter stator windings, thus causing no current to flow.
  - C. Unequal to the voltages induced in the transmitter stator windings, thus causing no current to flow.
  - D. Unequal to the voltages induced in the transmitter stator windings, thus causing current to flow.
6. If a transmitter-differential receiver-transmitter synchro system were connected for addition, and if the rotors of both transmitters were rotated  $60^\circ$  in the clockwise direction, the differential receiver would:
- A. Rotate  $120^\circ$  in the clockwise direction.
  - B. Rotate  $120^\circ$  in the counterclockwise direction.
  - C. Rotate  $60^\circ$  in the clockwise direction.
  - D. Remain at  $0^\circ$ .
7. The stator currents of a control transformer are determined by:
- A. The voltages applied to them.
  - B. The amount of rotor displacement from its reference.
  - C. The direction of rotor displacement from its reference.
  - D. All of the above.
8. Which of the following servomechanisms provide the greatest amount of system control?
- A. Closed-loop servomechanism.
  - B. Open-loop servomechanism.
  - C. High-gain amplification servomechanism.
  - D. Low-gain amplification servomechanism.

9. In a closed-loop servomechanism using a follow-up control transformer, the voltage from the follow-up control transformer is:
- A. In phase with the voltage generated in the input section.
  - B.  $90^\circ$  out of phase with the voltage generated in the input section.
  - C.  $180^\circ$  out of phase with the voltage generated in the input section.
  - D.  $120^\circ$  out of phase with the voltage generated in the input section due to one stator winding not being used.
10. A 10-bit DAC has an output range of from 0 to 5 volts. Its resolution is:
- A. .097%
  - B. .00097%
  - C. 4.8 mV
  - D. Both A and C are correct.
11. Most commercial DACs are:
- A. Binary weighted ladder DACs.
  - B. R-2R ladder DACs.
  - C. Multiplying DACs.
  - D. Successive approximation DACs.
12. Double buffering is required when an 8-bit microprocessor is interfaced to:
- A. 8-bit DACs.
  - B. 6-bit DACs.
  - C. 10-bit DACs.
  - D. Double buffering is never required when you are interfacing to DACs.
13. Which of the following would **not** be a practical DAC application?
- A. Motor control.
  - B. Waveform generation.
  - C. Temperature sensing.
  - D. Thermostat control.

14. Which type of ADC has the best overall qualities, considering accuracy, speed, and cost?
- A. Ramp generation.
  - B. Successive approximation.
  - C. Integration.
  - D. Dual slope.
15. When must input latching be provided as part of the PIA/ADC interface?
- A. Always.
  - B. Never.
  - C. With ADCs less than 8-bits.
  - D. With ADCs more than 8-bits.
16. The two major control lines on a ADC device are:
- A. DOC and END.
  - B. START and EOC.
  - C. BUSY and Data Valid.
  - D. RUN and Initiate Conversion.



## EXAMINATION ANSWERS

For your convenience, the page where the correct answer can be found is shown following the answer.

1. C — Synchro systems provide a higher torque output. [8-6]
2. B — Control Transformer (CT). [8-12]
3. A — Differential synchro transmitter. [8-10]
4. B — 36.8 V, 13.5 V, 50.2 V. [8-14,15]
5. D — Unequal to the voltage induced in the transmitter stator windings, thus causing current to flow. [8-19]
6. D — Remain at 0°. [8-24]
7. A — The voltage applied to them. [8-29]
8. A — Closed-loop servomechanism. [8-33]
9. C — 180° out of phase with the voltage generated in the input section. [8-38]
10. D — Both A and C are correct. [8-42]
11. B — R-2R ladder DACs. [8-46]
12. C — 10-bit DACs. [8-49]
13. C — Temperature sensing. [8-51,52,53]

14. B — Successive approximation. [8-58,60]
15. A — Always. [8-62]
16. B — START and EOC. [8-61]

*Unit 9*

**VOICE SYNTHESIS**

## CONTENTS

Introduction .....	9-3
Unit Objectives .....	9-5
Unit Activity Guide .....	9-6
How Do We Speak? .....	9-7
Fundamentals of Speech .....	9-20
Looking at the Waveforms .....	9-29
Producing a Phoneme Electronically .....	9-36
Using a Phoneme Synthesizer .....	9-45
Programming the PSS .....	9-50
Naturalness and Phoneme Phrases .....	9-64
Experiment .....	9-74
Unit Examination .....	9-75
Examination Answers .....	9-79

## INTRODUCTION

All around you, equipment is using synthesized voices to convey warnings and information. In some automobiles, a voice warns you to check your fuel supply or turn off your lights when you leave the car. Microwave ovens tell you when food is ready to be served. Voice communication by man-made means is not a new vogue; in fact, many robots, especially display and hobbyist robots, have had the capability of speech for several years. However, it was usually a pre-taped recording or some type of radio transmission. In comparison, several of today's robots are using synthesized speech to convey messages to their operators. These messages may contain information about the robot's own needs, or they may convey information pertinent to the work the robot is performing. Whatever the case, voice technology soon will be the norm rather than the exception.

But before you can add an electronic device to your robot, you must decide what it is that you want the device to do. You should already be familiar with most of your robot's capabilities; now you must learn the synthesizer's nature and how it will interact with your robot. The more you know about the hardware, the easier it will be to coordinate the system activities.

Naturally, the main point of having a synthesizer connected to your robot is to give the robot a voice. But how should it talk? What sort of speech should it have? Is mechanical-sounding speech adequate, or will you need a synthesizer whose speech could pass for a human's? Will it say only a few words, or will you expect the robot to carry on long conversations? The speech will be artificial speech, of course, but only in the sense that synthesizers aren't human speakers. Other than this unavoidable circumstance, you might want it to be as much like human speech as possible. There are conditions, however, under which you will insist that your robot sound like a robot — mechanical and monotonic. You have to understand how you want the speech to work with the rest of the robot's operations. Do you want speech to humanize the robot? Are the spoken messages going to relate to the robot's own needs, to its surrounding environment, or both?



When you complete this unit, you will have gained the knowledge to make these decisions. Although this unit focuses on the ways you can produce speech electronically, the complete study of speech synthesis includes mechanical synthesizers, as well as electronic devices. Now is your chance to become fully conversant with the synthesis revolution. First, you will learn some of the common techniques of speech synthesis from the theory presented in the first portion of this unit. Then you can use your Robot Trainer to perform the experiments and actually make your robot talk. This unit gives you the essential ingredients of a synthesizer so you can perform these experiments. Through this, you will learn about this exciting and important segment of technology. It takes a combination of experience and theoretical know-how to truly grasp the implications and applications of voice synthesis.

## UNIT OBJECTIVES

When you have completed this unit, you should be able to:

1. Define the following terms:  
    Phone.  
    Allophone.  
    Phoneme.
2. Give an example of the following sounds:  
    A sibilant.  
    A nasal.  
    A fricative.  
    Voiced.  
    A stop.
3. Explain how acoustically changing the pharynx will vary its resonant frequency.
4. Describe the basic difference between stored-word and stored-phoneme speech synthesis.
5. Identify the characteristic frequencies of the human voice.
6. Describe a phoneme spectrograph.
7. Identify the basic components of a phoneme speech synthesizer circuit.
8. Write a program to cause the phoneme speech synthesizer to "speak."
9. Explain how inflection and intonation can be used to place emphasis on specific words or phrases.
10. State the fundamentals of the three step phoneme concatenation process.
11. List five guidelines to successful inflection programming.
12. State the purpose of the intonation bar graph.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "How Do We Speak?"	_____
<input type="checkbox"/> Read "Fundamentals of Speech."	_____
<input type="checkbox"/> Answer Programmed Review Questions 1–7.	_____
<input type="checkbox"/> Read "Looking at the Waveforms."	_____
<input type="checkbox"/> Answer Programmed Review Questions 8–12.	_____
<input type="checkbox"/> Read "Producing a Phoneme Electronically."	_____
<input type="checkbox"/> Answer Programmed Review Questions 13–19.	_____
<input type="checkbox"/> Read "Using a Phoneme Synthesizer."	_____
<input type="checkbox"/> Answer Programmed Review Questions 20–25.	_____
<input type="checkbox"/> Read "Programming the PSS."	_____
<input type="checkbox"/> Answer Programmed Review Questions 26–29.	_____
<input type="checkbox"/> Read "Naturalness and Phoneme Phrases."	_____
<input type="checkbox"/> Answer Programmed Review Questions 30–36.	_____
<input type="checkbox"/> Perform Experiment 15.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## HOW DO WE SPEAK?

Speech is such a natural process that you seldom think about how you talk, but it is a complex mechanical process. It involves the continuous interaction of the lungs, windpipe, vocal cords, throat, nose, and mouth. For each sound that you make, your body coordinates these vocal organs, putting them in some precise combination of locations and motion. Your brain choreographs the sophisticated actions that produce speech; and, you do it without even thinking about it consciously!

Speech begins with an air stream that you force out of your lungs. Your lungs hold about three quarts of air, and your diaphragm can push out controlled amounts of that air to make just the right amount available at the right time. The amount of air, and the intensity with which you let it pass through your windpipe (trachea), determines how loud you will speak. A whisper uses very little air, while a yell requires quite a bit. This controlled air stream isn't speech, but it does act as the energy source for speech sounds.

We say that the air stream is an energy source because you don't normally hear air that is simply exhaled. If you hear anything at all, it is just the rushy sound of breathing. But when the air stream vibrates, you can hear it. When the vibrations are strong enough to vibrate the air mass outside your body, other people can hear it. The vibrations are audio frequencies. And, if the sounds you are making have some meaning to your listener, then you are speaking.

## The Larynx

The job of vibrating the air stream belongs to your vocal cords. However, as shown in Figure 9-1, your vocal cords aren't actually cords at all. They are folds of ligaments that stretch from front to back across your windpipe, controlling the air stream's flow like a trap door. You'll find this trap door in a set of cartilages, called the **larynx**, located about half-way up the front of your neck. The larynx, or Adam's apple, is so vital to the production of speech that it is sometimes called the voice box. Of course it isn't a box at all, but it is the starting point for the addition of information to the air stream from the windpipe.

Using the vocal cords, you can completely or partially block the air stream's flow — and you control the degree and speed of the blocking action by altering the tension of the ligaments. Changing the tension changes the size of the **glottis**, or opening in the vocal cords. You can open the glottis as much as half an inch, or close it off completely.

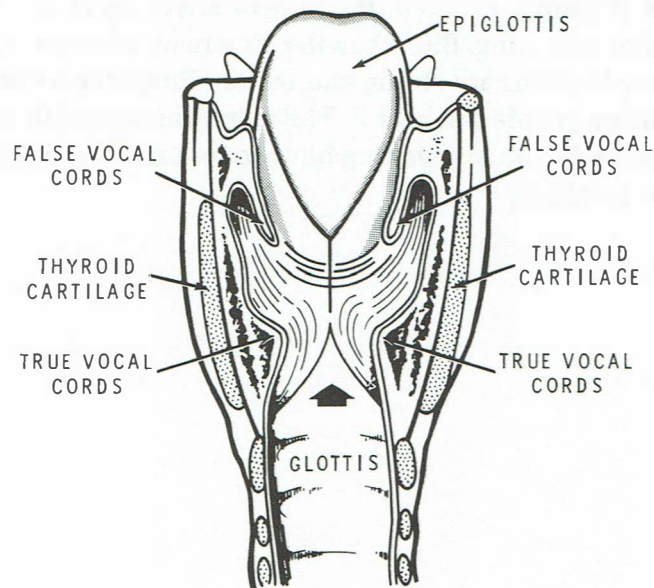


Figure 9-1

The larynx.



Opening and closing the glottis rapidly turns the steady stream of air into short bursts, or puffs. You hear these bursts as a buzzing sound. The faster you open and close the vocal cords, the higher the frequency of the buzz that you produce. In normal speech you'll vary the frequency of that buzz from 60 hertz to 350 hertz. The movement of the vocal cords to produce speech is called **voicing**.

When you use the vocal cords this way, the noises that you make are called, naturally enough, voiced sounds. Voiced sounds constitute only one category of all of the speech sounds you can make, but it is a large category. All of the vowels and some consonants are voiced sounds.

You can test out the action of the vocal cords for yourself by putting your fingers on your Adam's apple and saying nearly any sentence. As you say voiced sounds, you'll feel the buzz with your fingertips. The sounds begin at the glottis. However, when you say voiceless sounds, such as the "h" in "hello," you won't feel the vibration.

The windpipe, or **trachea**, and the **esophagus** meet at the larynx. Part of the larynx's job is to ensure that food goes down the esophagus to the stomach and that air goes up and down the trachea to the lungs. It acts as sort of a traffic cop at the end of the throat. Thus the larynx not only aides in speech, it prevents you from choking to death on your lunch.

When a person's larynx has to be surgically removed (such as when cancer has destroyed it) or has been paralyzed by disease or injury, normal speech becomes nearly impossible. Some people, however, can learn to get by with a trick called esophageal speech. This is speech that you generate by taking air into the stomach and expelling it in a way that it vibrates the entry folds of the esophagus; thus interrupting the air flow through the larynx. It isn't as effective as vocal-cord vibration and some people aren't able to learn esophageal speech at all. These people must resort to mechanical devices that are held to the trachea. The mechanical larynx actually vibrates the trachea, thus vibrating the air stream. But the speech produced has an artificial quality to it — more so than you'd usually want a synthesizer to have. You just don't get the true air bursts that opening and closing the glottis creates. But at least it is understandable speech that allows the person to communicate with a minimum of inconvenience and hardware.

## The Vocal Tract

But it isn't just the frequency of the buzz that distinguishes one voiced sound from another. Speech sounds are more than just a buzz. After it leaves the larynx, the air stream passes through the throat (**pharynx**), mouth, and nose. All of these change the sound slightly. These organs make up the speech apparatus known as the vocal tract. And the vocal tract, shown in Figure 9-2, plays an important role in determining the final sound that is produced.

As the air stream passes through the pharynx and mouth, the natural resonance of these cavities modifies the vibration (buzzing) into a variety of sounds. By moving your tongue to various positions inside the mouth, opening or closing your teeth, raising or lowering the palate, or otherwise changing the shape and size of the vocal tract, you change its resonance. This, in turn, causes the air stream to be modified. This modification process is sometimes called modulation.

Engineers like to represent processes that take place in the world with models that they can experiment with and control under laboratory conditions. Then they are able to make guesses about the effects of circuit or vocabulary changes and test them out with the model. Therefore, they tend to think of the vocal tract as a pair of adjustable resonant tubes arranged in series rather than as a throat, mouth, and nose. The air stream from the larynx acts as a quasi-periodic energy source that excites the cavities. This model is much easier to draw than the entire vocal tract, and proves much easier to represent with equations.

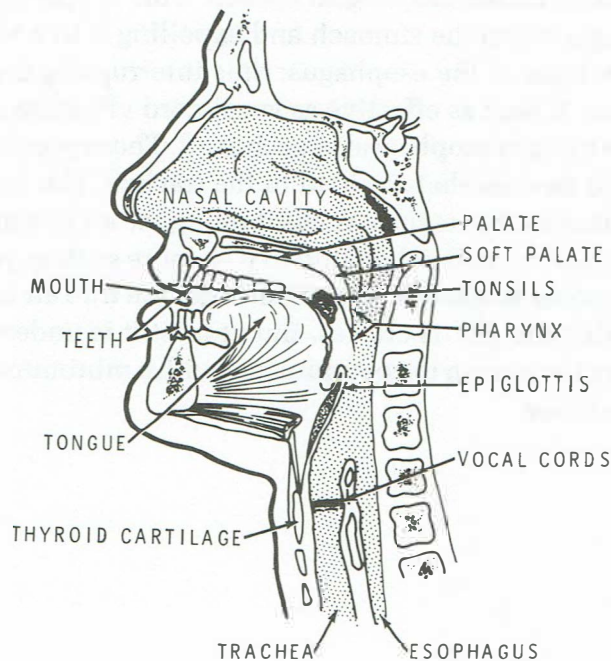


Figure 9-2

The vocal tract.

One successful model, shown in Figure 9-3, uses the equations for a mechanical system, called a double Helmholtz resonator, named for its inventor, the German professor of anatomy and physiology, Hermann Ludwig Ferdinand von Helmholtz. Helmholtz studied both tube resonance and the human vocal tract to create his model. The loosely coupled tubes must be adjustable; changing the position of the tongue, opening the nasal passages to the pharynx or otherwise altering the size or shape of the oral and nasal cavities divides up the cavities into different-sized chambers, thus changing their resonance.

A tube's resonant frequency is the frequency that echoes naturally inside it. An echo reinforces the signal, making the signal stronger, or higher in amplitude at that frequency than at other frequencies. The resonant frequency depends on the size and shape of the cavity, or tube, since these things determine which frequencies echo the strongest. The smaller the tube is, the higher is its resonant frequency.

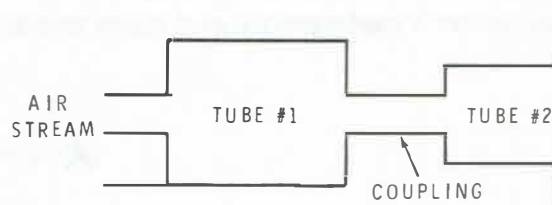


Figure 9-3

A two-tube model.

If you excite the tubes with a signal that contains all of the frequencies from 500 to 5000 Hz in equal amplitude, and, if the tubes resonate at 1000 Hz, the signal that comes out will have an amplitude peak at 1000 Hz as shown in Figure 9-4. A smaller tube will have a peak at a higher frequency, and a larger tube has a peak at a lower frequency. However, the double Helmholtz resonator has two resonant cavities. Therefore, it echoes, or resonates, at more than one frequency. Thus, the signal has several such peaks.

Not only can designers use this model to successfully create both mechanical and electronic speech synthesizers, but medical diagnosticians and therapists can use the model to gain an understanding of speech defects. This understanding can lead to ideas for physical therapy that will help the individual overcome the defect. After all, if the model explains the actions of your body in producing speech, it should illustrate what is going wrong as well as what is working properly.

Not only does understanding the speech process aid engineers when they are designing synthesizers, but it helps you to use synthesizers properly too. Remember that the model provides an affect that your synthesizer can mimic. If you understand the effect itself, you are better equipped to appraise the synthesizer's performance and make minor changes in the way you use it.

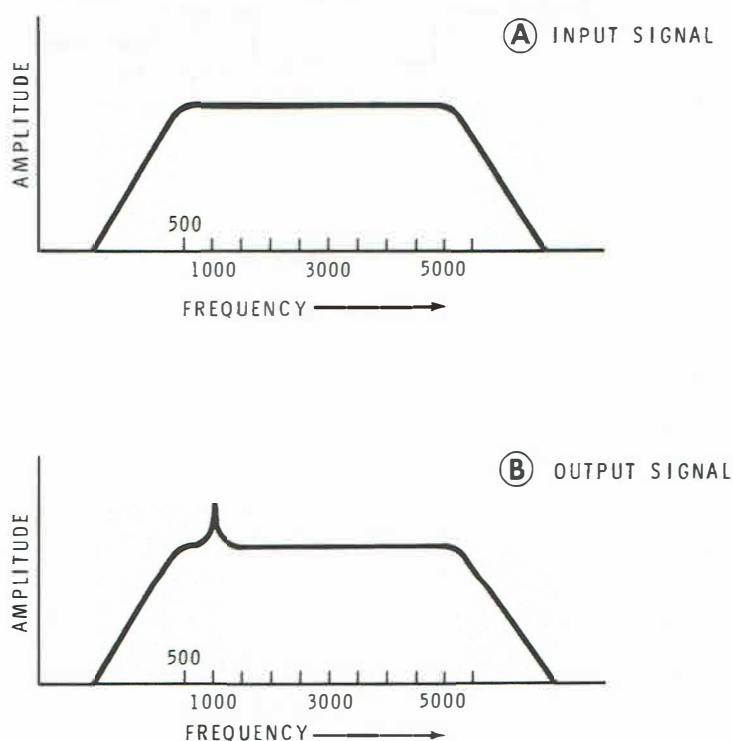


Figure 9-4

Flat input — output peaked at 1 kHz.



The synthesizer won't necessarily copy the human speech process exactly. As you'll see later, there are several approaches to speech synthesis. Yet, the 2-tube resonance model explains the physical effects of speech even if it falls short of explaining the entire speech process. Copying the model gives good results; and, if your synthesizer does use that approach, that's all you care about. In fact, in 1949, researchers at Bell Labs directly converted a limited 2-tube resonance model's acoustic-resonance characteristics into their electrical equivalents, shown in the schematic of Figure 9-5. This was one of the first electrical synthesizers. They called it the Electrical Vocal Tract, or EVT. The EVT had definite limitations, however. It could only make English vowel sounds, and it didn't always do them too well. But the experiment proved that the model was, within certain limits, correct.

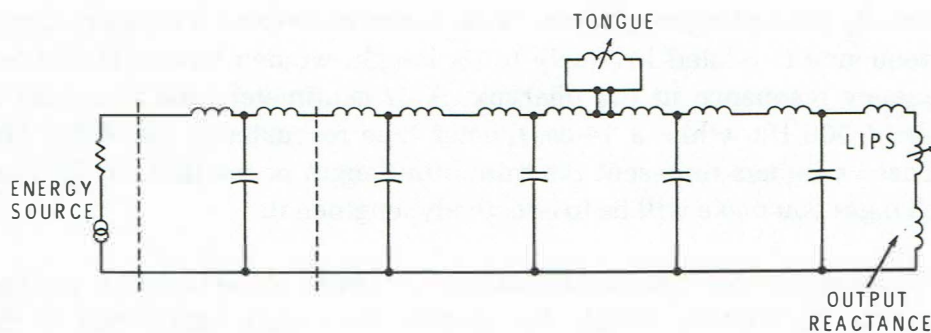


Figure 9-5

The EVT schematic.

The vocal tract has reasonably direct counterparts to the elements of the model, but you have to understand that speech is a dynamic process. The constantly moving tract might look like one form of the model for the time it takes to make one speech sound and totally different while it makes another.

Engineers will tell you that, in the model, a vibrating air stream excites the two resonant tubes. This just means that the vibrating air stream is the energy source for the speech. The tubes don't act on that air stream, they **react** to it, adding their own resonance characteristics to the signal.



In exactly the same way, the puffs of air from the larynx excite the harmonic resonance of the vocal tract. The resonant frequency of the tubes in the model, or the resonance of the two major sections of the vocal tract in people, determines which of the various possible sounds are produced. The actual changes in the speech, due to the resonance of the vocal tract, represent what engineers call the tract's transfer function. A transfer function is just a mathematical method of representing the changes that a circuit, or resonant cavity, makes to the input signal.

The pharynx, or throat, is the first resonant section of the vocal tract. It begins at the glottis and ends at the back of the mouth. You don't normally change this tube's characteristics while you are speaking. Its major factor in affecting speech is its length, and you can't change that.

Men usually have a pharynx that is about 17 centimeters long; a woman usually has a pharynx of about 14 centimeters. Because a tube's resonant frequency is related inversely to its length, women have a higher-frequency resonance in the pharynx. A 17-centimeter tube resonates at about 500 Hz, while a 14-centimeter tube resonates at about 607 Hz. These numbers represent the minimum length of the first cavity. Any changes you make will be to effectively lengthen it.

Although you can't physically change the length of the pharynx, you can change its acoustic length. By opening the mouth completely to the pharynx, the two cavities can act as one tube. This tends to be the case when you make sonorants, those full-sounding vowels, such as /a/.

Certain speech sounds, called nasals, are made by opening the nasal cavity to the pharynx during speech. You've heard people who "talk through their noses." Unless the person has problems with adenoids, people whose speech is extremely nasal haven't learned to control the soft palate, or **velum**, a muscle which separates the mouth from the nasal cavity. The soft palate is shown in Figure 9-6.

The nasal cavity's four inches of space dramatically alters the quality of the speech — not always for the bad. Some speech sounds depend on the nasal cavity. Without the nasal murmur, as it's called, you'd have a hard time making either an "m" or an "n." In fact, you'd sound exactly like someone who's head is stuffed up due to a cold. They have lost the use, temporarily at least, of the nasal cavity. The nasal cavity isn't the second of the two tubes in the model, however. Instead, it acts as a modifier, or side resonator that adds its own resonance characteristics to those of the true second tube — the mouth.

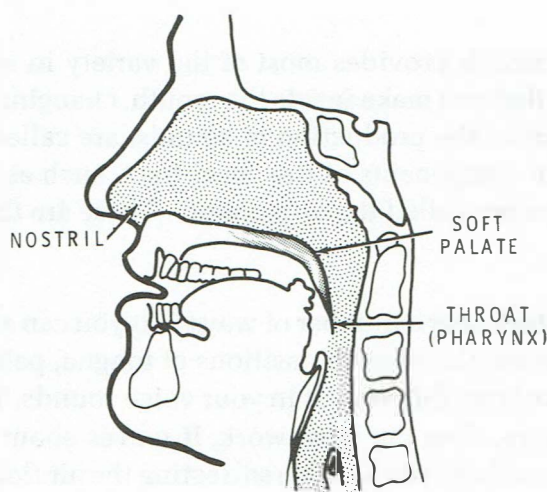


Figure 9-6

Soft palate.

## The Articulators

Although the mouth can sometimes act as part of the pharynx, such as when you use it to make the pharynx acoustically longer to produce open sounds, it also provides the model's second resonant tube.

But how can the mouth be both part of the first, or pharyngeal, cavity and the second cavity too? Well, the tongue helps out a lot, often dividing the mouth into two acoustic chambers. Sometimes the second tube might only be the space from where the tongue touches the hard palate to the lips, while the rest of the mouth works with the pharynx to produce an extremely long first cavity. You make this configuration every time you make the /i/ sounds. The first cavity then has a low resonant frequency because you have lengthened it acoustically and larger cavities resonate at lower frequencies: while the tiny second cavity resonates at a relatively high frequency. This shows how flexible the mouth can be in producing sounds.

In fact, the mouth provides most of the variety in speech sounds. The movements that you make inside the mouth, changing its shape and resonance to control the production of sounds, are called **articulation**. The most flexible components of the vocal tract, such as the tongue, palate, teeth and lips are called the articulators. These are the active speech organs.

The articulators offer a number of ways that you can alter the air stream's flow. By moving the relative positions of tongue, palate, lips, and teeth, you can make large differences in your voice sounds. The tongue, of all of the articulators, does the most work. It moves about the inside of your mouth constantly as you speak, redirecting the air flow and changing the mouth's acoustic characteristics.

There are two factors, or manners, that determine the effect of an articulation: the amount you constrict the airflow through the vocal tract and the point at which you make the constriction. This second factor is called the **place of articulation**. If you excite the vocal tract with a particular air stream (assume for a moment that you can measure the characteristics of the air stream coming out of the larynx and reproduce it exactly) when you put the tongue near the hard palate, therefore, blocking part of the airflow through the mouth, you'll make what is called a close vowel such as the /ee/ sound in "beat."

Put your tongue at the bottom of your mouth while keeping everything else constant (especially our hypothetically reproduceable air stream) and, since the passage is relatively unobstructed, you'll make an open vowel sound like the /a/ in bat. Thus, the difference between open and close vowels is the place of articulation. All vowels are relatively open vocal sounds, even the /oi/ in "toil" or the /u/ in "put." The close vowels, which you form by constricting the air stream, are only close when they are compared to the wide open vowels, such as /a/. The /e/ sound, for example, is wide open when you compare it to a consonant, such as /t/.

## Voiceless Sounds

Some sounds depend entirely on the actions of the articulators to exist at all. No puffs of air, no vibrating air stream acts as their energy source. These sounds are called voiceless, or unvoiced. No, this term doesn't contradict itself. Since, in speech terminology, the action of the vocal cords is termed voicing, when you don't vibrate the air stream but still articulate speech sounds, the sounds are called voiceless. They do have to be articulated, however. Even if you learned to make vowel sounds by beating two rocks together, they would still be voiced sounds.

But if you don't use the vocal cords to vibrate the air stream, how do you make the silent air stream into audible speech sounds? Actually, there are several ways of doing that. Therefore, there are several types of voiceless sounds, each characterized to some degree by the manner by which you create it.

You can, for example, form a constriction at some point in the vocal tract, say with the lips and teeth, and force air through it at high enough speeds that the friction of the air stream will produce turbulence. This turbulence makes fricative sounds, such as an /f/. The word **fricative** merely implies that the sound was "made by friction."

By changing the place of articulation slightly, the fricative /f/ becomes the fricative /s/. The /s/ makes a higher frequency sound than the /f/ and is classified as a sibilant. The term **sibilant** comes from the word **sibilate**, which means simply "to give a hissing sound." Both the /f/ and /s/ are consonants and fricatives, however. They are both consonants because that is their linguistic role; they are both fricatives because they are made with friction (turbulence); the /s/ is a sibilant because it gives a hissing sound. Obviously, there are many ways to classify a sound.



If you close the vocal tract off and completely block the air flow, by closing your lips tightly for example, you are making a **stop** sound, such as a /b/. Allow the air stream to build up pressure behind the closure; then open the block suddenly and you've made a specialized stop called a plosive. The plosive is usually followed by some fricative noise. Try holding your hand in front of a popping /p/, as in "pop." You should feel the high-pressure air following the sound.

Naturally, you can combine nearly any manner of articulation with either a voiced or unvoiced sound source. The fricative /f/ becomes a /v/, for example, when you simply add voicing. And, although you can make many types of sounds, they will all be either voiced (periodic, in the engineer's model) or unvoiced (turbulent, or white noise).

## Stringing Sounds Together

The vocal tract, in its role as a filter, must constantly be moving, changing its characteristics to handle any speech sound you want to make. And the articulators don't move quickly. Of course some combinations of sounds are harder to make than others. And, your mental computer must know ahead of time that when a /t/ is followed by an /h/ it isn't said the same way as when the /t/ is followed by an /o/.

Some letter combinations result in one of the sounds that you'd expect to hear becoming silent, such as the /e/ in "rude." You hear the presence of the /e/ — the /u/ gets modified, becomes longer — but you don't hear any /e/ sound.

In this example, the pronunciation changes because of language rules. Sometimes the changes aren't due to potential sound combinations within the word, but because of sounds in adjacent words. We don't say "bacon and eggs," for example. We say "bacon an eggs." In the case of "cheese and crackers" the phrase becomes "cheese 'n crackers." The "and" almost disappears completely.

This isn't because of some perverse set of rules or logic. It is just the way we do things in English. The language isn't an absolute; it evolves, changing to suit its speakers. And many changes that the speakers choose to make are just because some sound combinations are hard to say. Language is a tool. A tool that we continually modify to make easier to use.



Each speech sound's production is affected by the simple mechanics of moving the articulators. The articulation of one sound can be impacted by the articulation of a previous sound. The articulators not only move slowly, but some of them move slower than others. The soft palate, for example, takes longer to get into its proper position than the tongue does. If you want to say two sounds in a row that require the soft palate to be at the extremes of its possible positions in the mouth, then you must either speak with great deliberation to give the soft palate time to move or accept some slurring of your speech. The term slurring is proper, too. The effects of having to move the articulators to say one sound slop over onto the production of the next. The movements linger. The articulators have inertia.

This influence of one speech sound on another, due simply to the lingering effects of articulation, is called **coarticulation**. You can see that stringing speech sounds together always changes pronunciation to some degree. Because coarticulatory effects are produced by the mechanics of speech rather than by sets of rules, it makes the speaking process decidedly individualistic. You will speak differently when you are sick than when you are well, differently when you are rested than tired. Anything that affects articulation, such as a toothache or a cold sore in your mouth, also changes the effects of coarticulation.

Although linguists consider coarticulation to be a kind of distortion of the speech signal, when you hear someone speak deliberately, carefully saying each speech sound as if it were the only one to be said, it sounds odd, to say the least. At best, you'd think the speaker was stilted; in some cases you might not even be able to understand what was being said. Our ears are trained to accept the slurring of speech — at least the effects of the most common sound combinations — as the way speech should sound. Theoretically perfect speech is rare, not normal. This makes it difficult to program a synthesizer from a textbook.

## FUNDAMENTALS OF SPEECH

We speak to each other with words, the smallest units of speech that have any meaning. Anything smaller than a word is simply a sound. Words are organized using sounds; and to understand this organization of sounds called speech, and to understand the basics of voice synthesis, you need to understand the sounds themselves.

### Naming Sounds

To make it easier to talk about speech sounds, linguists have given names to speech sounds and developed a method of representing sounds and parts of sounds. In fact, there is a special branch of linguistics, called phonetics, that deals with the task of identifying and naming speech sounds. Phoneticians have developed a way of writing down a sound using special symbols and spellings. Using conventional spellings causes more confusion than clarification of speech habits.

Some of the names that phoneticians use explain themselves. Nasals, for example, are nothing more than sounds that have a nasal quality due to the soft palate located at the opening of the nasal cavity to the pharynx. Terms like fricative and sibilant, as you've seen, get their names from the way they are produced. Other terms aren't quite as easy to figure out, however.

## Phones

The term “phone” comes from the Greek language and means, simply, an articulated sound. The term articulation refers only to sounds that are produced by your vocal tract. If you clap your hands, you are making a sound, but not speaking. Handclaps aren’t speech sounds, not in English at least, and linguists restrict their study to speech sounds. Beyond this functional description, however, phones aren’t very well defined.

**Phones** act as the building blocks for speech. You concatenate, or string together, phones to produce speech. But the location of a phone in a word, or group of words, can alter the way that you say it. This might be due to coarticulation, rule, or habit. As a result, you can’t easily define some set of phones and say that by combining these elements according to some set of rules, you can produce speech. It’s almost impossible to capture all of the special cases. This is partly because the effects of coarticulation aren’t always as predictable as they might seem. It is also partly because of anticipation. You change the way you say a sound because of the sound following, either to make them both easier to say, or just to make them sound better.

You say the /p/ in “pan,” for example, differently than you would say the /p/ in the word “pin.” You anticipate having to say either the “an” or “in” and modify the way you say the /p/ to make the next phone easier to say.

This is the type of change that means a great deal to linguists. By categorizing all of the changes that come about, all of the combinations and their effects, they get a clearer idea of the mechanics and physics of speech. As you can see, the job is enormous.

## Phonemes and Allophones

When you change a phone, even slightly, it gets a new name. A phone that is modified because of coarticulation or any other factors is called an **allophone**. Some allophones sound quite different than the root phone. Although in the case of the different-sounding “p”s in “pin” and “pan,” it’s fairly obvious that they still function the same. However, consider the “p” in “phonetics.” It has a sound that would make you think it came from an /f/ rather than a /p/.

When you gather all of the phones or allophones that have the same function together under one label, such as classifying both of the two similar-sounding /p/s mentioned above as variations of the constant /p/, but the “p” from “phonetics” as a different unit, you have collected and classified a set of sounds called **phonemes**. Thus, phonemes are nothing more than groups of phones, or allophones, that all have the same linguistic meaning. Phonemes are the sounds that let a listener differentiate one word from another. When you “hear” the difference between words, you hear phonemes. Phones are sounds; phonemes are elements of speech — subsets of words. To the linguist, allophones are “context-dependent alternative phoneme forms.” That means that allophones are the way you modify a basic sound to suit a given word and the sounds around it.

A second group of linguists called phonemicians (as opposed to phoneticians) study the grouping of sounds (phones) into phonemes. This work proves vital to understand speech because phonemes are a language’s smallest fundamental units.

Each language has its own number of phonemes. It has exactly the number needed to make all of its speech sounds and no extras. This is why one language sounds different than another. Known languages contain from as few as two vowel phonemes to as many as 12 and from a dozen or so consonant phonemes to as many as 70. In English, we have about 40 phonemes in two categories: vowels and consonants. The linguistic experts can’t agree on an exact number, however.

One problem with figuring out how many phonemes any given language has is deciding what constitutes a standard pronunciation of that language. Within each language are dialects. These are nothing more than collections of speech habits that are common to a group of people. We have very descriptive terms for dialects. We talk about the Irish brogue, or the southern drawl. These terms actually describe the dialect's idioms and not the dialect itself. An idiom is just an irregular, or different, way of speaking, such as saying "Ya'll" for "You all."

Thus, a dialect isn't much more than a specialized way of speaking a language. Yet, the dialect, in part, determines how many linguistic units, or phonemes, you need to describe the speech. The more dialects a language has, the more varied they become, the larger the set of phonemes needed to describe the language. Consider a dialect that never uses the consonant /v/, for example. A linguistic observer would say that the language didn't have that phoneme. If it turned out that another dialect of the same language used the /v/, however, but didn't use some other phoneme that was used in the first dialect, you would have two phonemes that were unique to particular dialects, but still needed to describe the language.

One practical example is the Great American dialect, which is spoken in midwestern and western United States. This dialect uses 16 vowel phonemes and 22 consonant phonemes. This is the dialect that you hear from trained speakers, such as television announcers, because it is considered by experts to be the easiest for everyone to understand. It is the common denominator for all United States listeners. You can be assured that of all dialects in this country, the Great American has the fewest irregularities, and no unusual phonemes. It is a bland and unconfusing subset of the speech habits of the people throughout the country.



Some irregularities aren't restricted to a particular dialect. Nearly everyone, for example, says "did you?" as "dija?" at least part of the time. And some modifications come from the message you are trying to get across. There is a great deal of information embedded in modifications in the long-term variations of pitch, intensity, and timing. These are called the **prosodic** features. You perceive prosodic features as stress and intonation. Prosodic features are difficult to measure and categorize, but they are vital in adding information to speech. Prosodics play a role both within a word and within a sentence. In a single word, everyone has heard some other speaker put the emphasis on the wrong syllable. The emphasis is just the prosodic quality of stress. The syllable is simply a single impulse from the vocal tract.

In a sentence, prosodics let you know when you're hearing a question (the stress at the end of the sentence gives it away), being urged to buy, or being warned of disaster. Prosodics make the difference between "He's coming for **dinner**?" and "**He's** coming for dinner?" You'll have to understand prosodics to make your synthesizer's speech sound natural.

And how do you represent all these sounds, with their variations, plus the prosodic information? Words should be fairly simple to represent accurately. There are 26 letters in the alphabet that we use to spell our words. Unfortunately, conventional spelling (**orthography**) doesn't provide much information about how a word is said or should be said. Orthography is simply a system of spelling that follows a set of rules or common usage, or both. Thus, although we have a visual method of representing words with letters in written English, it is a method that, unfortunately, might or might not relate to the way the word is said. "Way" and "Weigh," for example, are valid orthographic spellings of two distinct words. Yet there isn't much practical speech information contained in these representations. Nothing in the spelling tells you that you should say them identically. Experience is the only guide to pronunciation. If these problems don't seem big to you, just ask any schoolchild who is learning to read whether or not conventional spellings make sense.

A phonemic transcription gives you a better grasp of a word's speech sounds. It is a higher level of information representation that relates to the speech sounds involved in saying a word. This is a method of writing out a word in terms of the phonemes used in saying it. But, phonemes have many variations — they are collections of variations. Thus, although a phonemic transcription gives you a great deal of pronunciation information, to get a spelling that conveys something precise about a word's pronunciation, you need a transcription that shows the exact allophone. This is called a phonetic transcription, which is written in terms of phones. Figure 9-7, for example, shows the phonemic transcription for the words weight and water.

<u>Conventional Spelling</u>	<u>Phonemic Transcription</u>	<u>ARPABET VERSION</u>	<u>Allophonic Transcription</u>
weight	/w ɛ I t/	/WEHIHT/	[W ɛ I t;]
water	/w a t ɜ /	/WAATER/	[w a t ɝ]

Note: The slash indicates a phonemic transcription's word boundary; square brackets do the same for allophonic transcriptions.

Figure 9-7

Spelling to phonemic transcription to allophonic transcription.

A new set of symbols is necessary to represent either phonemes or allophones accurately. The alphabet has only 26 characters, and English has about 40 phonemes. As a result, you'll see symbols that combine Greek letters with our own English alphabet to cover all the phonemic and allophonic transcriptions. But these are a bit awkward. Standard computers in this country don't read Greek well, if at all. Only special computer keyboards will even have the symbols on them.

To make life a little easier for those who simply want some kind of a standard for representing speech in a computer, the Advanced Research Projects Agency for Speech Understanding Research (ARPA SUR) defined the phoneme set shown in Figure 9-8. This collection is called the ARPABET. These representations give you 14 vowel phonemes, 3 diphthongs (blended combinations of vowel sounds), and 22 consonant phonemes. The set provides both computer-readable symbols and standard linguistic notation. In addition to the symbols you see here, you can follow a vowel immediately with a stress assignment. This is a number from 0 to 3, with zero indicating no stress and three indicating very heavy stress. This provides some rudimentary prosodic information. Now you have a way of indicating speech patterns.

Although the ARPABET was developed with speech recognition work in mind, it does represent a determined effort to create some standards for describing speech. The true beauty of the ARPABET choices is that the collection not only describes a useful set of phonemes, it also includes a notation that is just right for computers. By using one- and two-letter representations from the computer-readable ASCII alphabet, you get enough symbols to handle all the ARPABET's units. This opens the door for a number of computer scientists to work in the field with a standardized set of phonemes and a standard phoneme notation without having to get involved in a detailed study of linguistics. Although ARPABET won't suit every researcher's needs, it makes a nice starting place for standardization.

Phoneme	Computer Representation		Example	Phoneme	Computer Representation		Example
	1-Character	2-Characters			1-Character	2-Characters	
i	i	IY	beat	p	p	P	pet
l	l	IH	bit	t	t	T	ten
e	e	EY	bait	k	k	K	kit
ɛ	E	EH	bet	b	b	B	bet
x	@	AE	bat	d	d	D	debt
α	a	AA	Bob	g	g	G	get
Λ	A	AH	but	h	h	HH	hat
ɔ	c	AO	bought	f	f	F	fat
o	o	OW	boat	ʌ	T	TH	thing
U	U	UH	book	s	s	S	sat
u	u	UW	boot	ʒ or ʃ	S	SH	shut
ɔ	x	AX	about	v	v	V	vat
ɪ	X	IX	roses	ð	D	DH	that
ɜ	R	ER	bird	z	z	Z	zoo
αU or αw	W	AW	down	ʒ or ʒ	Z	ZH	azure
ai or ay	Y	AY	buy	ʃ	C	CH	church
ɔɪ or oy	O	OY	boy	ʒ	J	JH	judge
y	y	Y	you	m	H	WH	which
w	w	W	wit	syl l, l	L	EL	battle
r	r	R	rent	syl m, m	M	EM	bottom
l	l	L	let	syl n, n	N	EN	button
m	m	M	met	flapped t, r	F	DX	batter
n	n	N	net	glottal stop.?	Q	Q	
ŋ	G	NX	sing	Silence	—	—	
				non-speech Segment	!	!	laugh, etc.

AUXILIARY SYMBOLS (1- AND 2-CHARACTER CODES ARE IDENTICAL)			
Symbol	Meaning	Symbol	Meaning
+	Morpheme boundary	: 3 or ,	Fall-rise or non-term juncture
/	Word boundary	* **	Comment (anything except * or **)
#	Utterance boundary	' '	Apos.-surround special symbol in comment
:	Tone group boundary	( )	Phoneme class information
: 1 or .	Falling juncture	< >	Phonetic or allophonic escape
: 2 or ?	Rising juncture		

Figure 9-8

The ARPABET alphabet.

## Programmed Review

1.	A speech sound is said to be a _____ when it is produced by friction.
2.	(fricative) A _____ is an articulated sound that is used as a building block for speech.
3.	(phone) A phone that has been modified is referred to as an _____.
4.	(allophone) A set of phones or allophones are grouped together to form _____, which are sounds that distinguish one word from another.
5.	(phonemes) There are about _____ different phonemes in the English language.
6.	(40) Stress and intonation are called _____ features and can be caused by variation in pitch, intensity, and timing.
7.	(prosodic) The _____ table allows computer scientists to work in the field of synthesized voice without getting involved in a detailed study of linguistics.
(ARPABET)	



## LOOKING AT THE WAVEFORMS

One of the best ways to understand what a speech sound is all about is to take a close look at its waveform. You'd expect that you'd measure its pitch, loudness, quality (timbre), and duration. But these are perceptual qualities of the speech signal, qualities that you feel and hear. Instead, you have to translate them into characteristics that better suit synthesizer evaluation — acoustic characteristics. You'll need to evaluate the speech signal in terms of measurable qualities such as intensity, fundamental frequency, and spectral content.

### What You See . . .

A speech signal's waveform quickly shows characteristics that aren't readily evident when you measure individual acoustic parameters. When you examine the wave's acoustic parameters, you find out about such things as the peak amplitude, the number of zero crossings in a given period, the signal's total energy, and its relative intensity. When you look at the waveform, you can immediately see the interaction of amplitude and frequency over the entire time period of the sound.

Speech has a complex waveform, however; just looking at a random sample displayed on an oscilloscope might put you into what computer scientists call **data overload**. This means that the unqualified display gives you more information than you can use. Much data about the speech signal gets lost. The patterns are too complicated to represent anything.

The search for a useful way to display and examine the speech waveform has led to the design of specialized equipment. In the 1930s and 1940s, researchers at Bell Telephone Laboratories in Murray Hill, NJ, developed a new piece of equipment called the sound spectrograph. This device produced what Bell researchers, in 1947, termed “visible speech.” The spectrograph displayed speech energy, with time on the horizontal axis, frequency on the vertical axis, and indicated relative intensity by the degree of darkness of the graph. Figure 9-9 illustrates the type of display you’ll get with a spectrograph for a male speaker saying the ARPABET vowel /i/, as in “eat.”

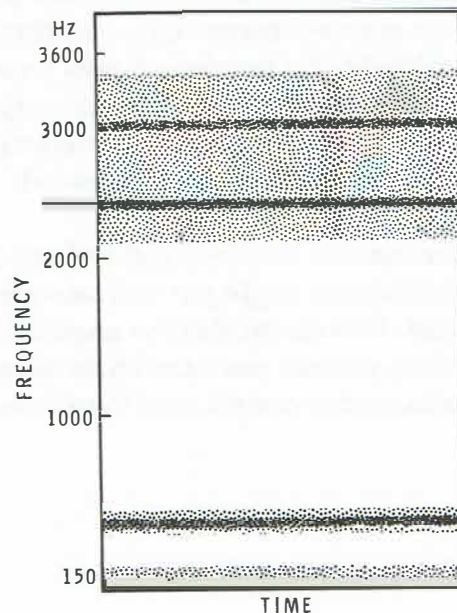


Figure 9-9

A spectrograph of /i/.

## Breaking Down the Frequency Domain

So, just what are the speech frequencies? You can find most of the information in the speech waveform within a frequency band of about 4 kHz. Many published spectrographs that provide important information, in fact, cover only the frequencies from 150 Hz to 3600 Hz, a 3450 Hz bandwidth. Commercial telephone systems typically transmit only the frequencies between 300 Hz and 3000 Hz — a bandwidth of only 2700 Hz. This is called the C Message band. Of course, anyone who has talked on telephone systems over long distance knows that some speech information does get lost when you reduce the bandwidth that far. If you want to be sure to capture all of the information in the speech signal, therefore, plan on using equipment designed to cover the frequencies from 0 to 5000 Hz.

The speech signal can contain many combinations of frequencies within this 5 kHz bandwidth. And the spectrograph provides valuable information concerning the relative intensity of these various frequencies. You can graph the relationship of a sound's intensity in several dimensions. First, however, you must understand the difference between the absolute intensity of the sound and the relative intensity of the frequencies within the sound.

The sound's total energy, or intensity, is the power per square unit of area in the acoustic wave. This is the power that gets transmitted along the acoustic wave. The standard measurement of intensity is the power sent through an area of one square centimeter that is placed at right angles to the direction that the sound wave is sent. This power is measured in watts.

The sound's relative intensity tells you how one sound compares to another. This comparison is made with the decibel scale. Decibels tell you ratios, not absolute values. A sound that is twice as intense as the reference (which might just be another signal) is 3 decibels (or dB) above that reference. A sound four times as intense is 6 decibels higher than its reference. As you can see, the relationship isn't linear — it's logarithmic. To calculate the relative intensity, you need some reference point. Then, to calculate the relative intensity in decibels, use the formula:

$$\text{dB} = 10\log_{10}(I/I_0)$$

where  $I$  is the intensity in watts of the reference signal and  $I_0$  is the new signal's intensity in watts. Another way to view intensity is to graph the intensity versus the signal's frequency for some point in time. This gives you the spectral envelope. Charting the spectrograph information that you already have, for the vowel /i/ in "eat," produces the chart shown in Figure 9-10. Of course, you can take the amplitude variations from this envelope and calculate the relative intensity in dB as before.

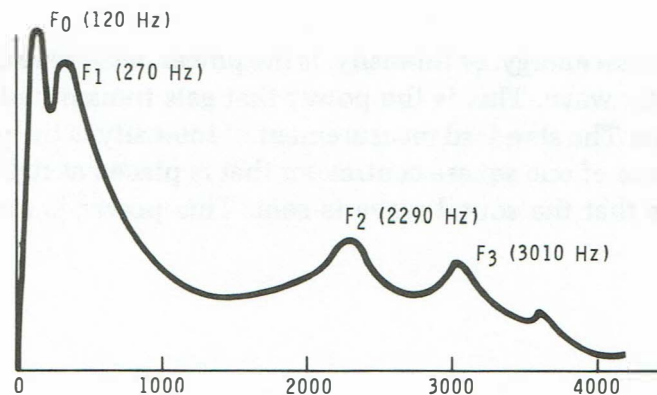


Figure 9-10

A spectral envelope of /i/.

Notice that the envelope has three peaks in the 150 Hz to 3600 Hz range. This is an extremely important characteristic of speech sounds — there isn't just one frequency that is interesting, but three or more. These peaks are the resonant frequencies of the vocal tract at the time the sound was made. Combine these resonant frequencies with the harmonics near them and you have a **formant**. Each speech sound will have a unique combination of these formants. They effectively describe the resonant frequencies of the vocal tract at that time. Our example speaker spoke with a fundamental frequency or pitch of 120 Hz, a fairly common pitch for male speakers. You'll sometimes see the envelope's fundamental frequency referred to as formant 0 (F0).

The first formant (F1) in our example is at 270 Hz; the second (F2) is at 2290 Hz; the third (F3) is at 3010 Hz. According to averages established by research at Bell Telephone Laboratories, if the speaker had been female, for the same vowel you'd expect a fundamental pitch of approximately 235 Hz, F1 should be 310 Hz, F2 should be 2790 Hz and F3 would be at 3310 Hz. For vowels and sonorants, these first three formants directly reflect the positions of the articulators. This helps investigators use formant analysis to diagnose many speech defects.

When you make an /i/ sound, you acoustically lengthen the pharynx. Now we have a tool to see if that really happens. A male speaker's pharynx should resonate at about 500 Hz — the natural resonant frequency for a 17-centimeter tube. The first formant in this example, however, is at 270 Hz. Somehow, probably by opening the pharynx to the mouth, the cavity was acoustically lengthened to resonate at a lower frequency. The second cavity should be small; just the distance from where the tongue constricts the airflow against the hard palate to the lips. A small cavity should have relatively high resonant frequency. The second formant is at 2290 Hz. The spectral information, therefore, agrees with what we know about the physics of speech.



And here is where therapists can use spectrographs. If you knew that an individual had trouble saying the /i/ sound, spectral analysis would show whether the problem was with the back cavity or front cavity. You could experiment with alternative methods of producing the correct formants, and thus the correct sound for /i/.

The lower-frequency formants are usually of a higher amplitude than the higher-frequency formants. When you listen, you focus on the lower-frequency formants. This is called the masking effect and it makes the lower frequencies much more important. This concept is important when you evaluate synthesizers. It means that when you listen to a synthesizer that has a bandwidth of 4 kHz compared to one with a bandwidth of 5 kHz, you won't hear as much difference as you might expect.

## Programmed Review

8.	The frequency range of the spectrograph is often only _____ to _____ Hertz.
9.	(150, 3600) The C Message bandwidth typically used by tele- phones is _____ to _____ Hertz.
10.	(300, 3000) The resonant frequencies and adjacent harmonics in a phoneme are called _____.
11.	(formants) Lengthening the pharynx causes it to resonate at a _____ frequency. (lower/higher)
12.	(lower) Because of the masking effect, _____ (lower/higher) frequencies are more important to the phoneme.
	(lower)

## PRODUCING A PHONEME ELECTRONICALLY

There are two primary methods of electronically producing synthesized speech — stored words and stored phonemes. Using the stored-word method, the entire vocabulary is fixed, therefore limited, and must be stored in either an external read-only-memory (ROM) or inside the speech chip itself. You may be able to request a specific vocabulary from the manufacturer. However, once it is stored, the system is limited to those words and to the way in which they were recorded. Many of the newer “talking” appliances use the stored-word method of synthesized speech to communicate with the user.

However, in the case of the ET-18 Robot Trainer, you may not always know in advance what you want it to say, and quite possibly, you will want to be able to vary its vocabulary. In both of these instances, you will need a synthesizer with much more flexibility. To meet these needs, the **phoneme speech synthesizer**, or PSS, is the ideal method of producing electronic speech. Because a robot may require an extensive vocabulary, and since memory is at a premium, we will limit our discussion to the phoneme speech synthesizer.

The idea behind phoneme synthesis is that you can store phonemes in memory just as easily as you can words. As you recall, phonemes are the building blocks of speech. Therefore to create speech, all you have to do is concatenate phonemes, which means to string them along in the correct sequence. The phoneme string is then applied to PSS which reproduces the sound. Thus, when compared to stored-word synthesizers, which store a limited number of words, the PSS can synthesize an infinite vocabulary using a minimum amount of memory.

## The Synthesizer Circuit

So what is a phoneme synthesizer? More importantly, how is it different from stored-vocabulary systems? First of all, you could use virtually any existing technique to store a highly flexible set of phonemes. Next, you would find that phoneme synthesizer circuit operations are really similar to those of stored-word speech chips.

For instance, the PSS contains:

- A parallel port that accepts phoneme commands.
- A phoneme look-up table.
- Articulation generators.
- Two sound sources.
- A digital filter.

To see how these work together to synthesize phonemes, examine the block diagram in Figure 9-11.

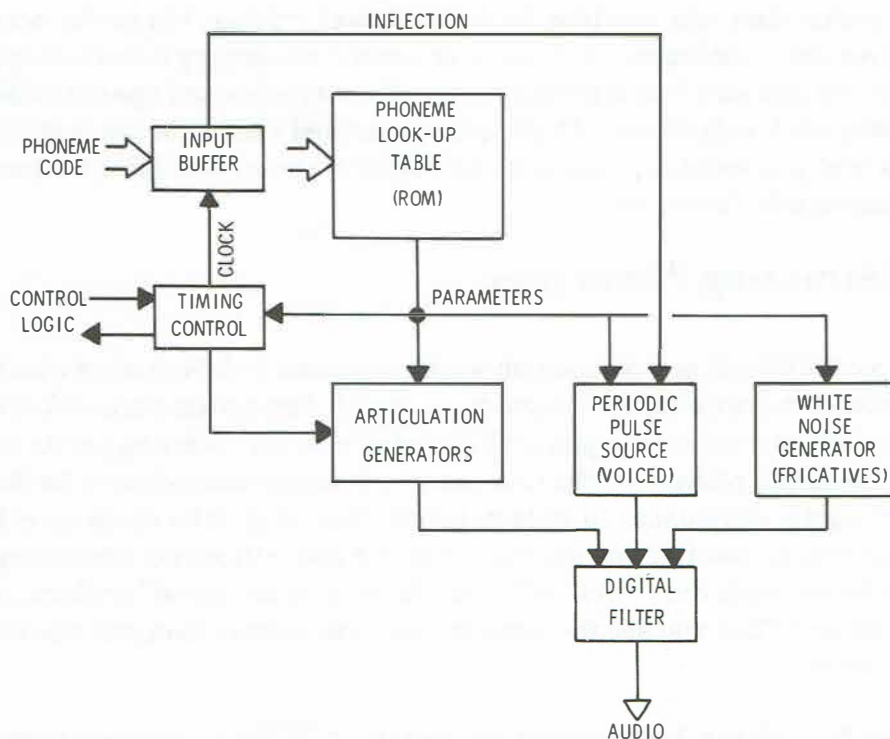


Figure 9-11

Block diagram of a typical phoneme speech synthesizer.

A digital code from the controlling processor indicates the phoneme to be produced. When it is to be a voiced phoneme, its pitch, or inflection level is also indicated. The synthesizer's look-up table, which is stored in ROM, translates the incoming phoneme code to a set of parameters, and then passes these parameters to the timing circuits and articulation generators. It is this translation process that makes phoneme synthesis different from other synthesizers.

The phoneme parameters provide control over the duration of the phoneme, the excitation of the basic sound generators, and the settings of the filters that model the human vocal tract. Each filter pole contributes part of the effect of the multiple resonant cavities found in the human vocal tract.

One major difference between the phoneme synthesizer and stored-word synthesizer is their quality of speech. Speech quality is largely due to the speed at which the synthesizer chip operates. The phoneme synthesizer is capable of producing continuous speech with a very low data rate, typically 70 bits per second; whereas the stored-word synthesizer, with a fixed vocabulary, operates at a data rate of 1200 to 2000 bits per second. The higher data rate provides for better speech quality, but at the same time requires more memory. Thus, if unlimited vocabulary is desired and processor memory is at a premium, phoneme synthesized speech is the obvious choice; however, if high quality, limited speech, is the requirement and processor memory is of no concern, stored-word synthesized speech may be the answer.

## Determining Phonemes

The most difficult part of using phoneme synthesis is determining which phonemes to string together to produce a word. Any systems or quick-reference tables used to help you pick phonemes are only starting points. In fact, the entire phoneme selection process is further complicated by the many subtle differences in speech habits. You must take these speech habits into account or you are sure to select many incorrect phonemes. Take for example the vowel "A". It can be long, as in "name" or short, as in "father." Thus you see the need for more phonemes than just one for each letter.

Figure 9-12 shows the phoneme list used with VOTRAX phoneme synthesizers. The list includes all the recognized phoneme sounds, plus a few special phonemes used for unusual sound combinations. A complete breakdown of these phonemes by basic sound characteristics is given in Figure 9-13.



Hexadecimal Phoneme Code	Phoneme Symbol	Duration (ms)	Example Word
00	EH3	59	jacket
01	EH2	71	enlist
02	EH1	121	heavy
03	PA0	47	no sound
04	DT	47	butter
05	A2	71	made
06	A1	103	made
07	ZH	90	azure
08	AH2	71	honest
09	I3	55	inhibit
0A	I2	80	inhibit
0B	I1	121	inhibit
0C	M	103	mat
0D	N	80	sun
0E	B	71	bag
0F	V	71	van
10	CH*	71	chip
11	SH	121	shop
12	Z	71	zoo
13	AW1	146	lawful
14	NG	121	thing
15	AH1	146	father
16	OO1	103	looking
17	OO	185	book
18	L	103	land
19	K	80	trick
1A	J*	47	judge
1B	H	71	hello
1C	G	71	get
1D	F	103	fast
1E	D	55	paid
1F	S	90	pass

Hexadecimal Phoneme Code	Phoneme Symbol	Duration (ms)	Example Word
20	A	185	day
21	AY	65	day
22	Y1	80	yard
23	UH3	47	mission
24	AH	250	mop
25	P	103	past
26	O	185	cold
27	I	185	pin
28	U	185	move
29	Y	103	any
2A	T	71	tap
2B	R	90	red
2C	E	185	meet
2D	W	80	win
2E	AE	185	dad
2F	AE1	103	after
30	AW2	90	salty
31	UH2	71	about
32	UH1	103	uncle
33	UH	185	cup
34	O2	80	for
35	O1	121	aboard
36	IU	59	you
37	U1	90	you
38	THV	80	the
39	TH	71	thin
3A	ER	146	bird
3B	EH	185	get
3C	E1	121	be
3D	AW	250	call
3E	PA1	185	no sound
3F	STOP	47	no sound

\*|T| must precede |CH| to produce CH sound.  
 |D| must precede |J| to produce J sound.

TABLE 1  
Figure 9-12

Phoneme chart (courtesy of VOTRAX).

Voiced					'Voiced' Fricative	'Voiced' Stop	Fricative Stop	Fricative	Nasal	No Sound
E	EH	AE	UH	OO1	Z	B	T	S	M	PA0
E1	EH1	AE1	UH1	R	ZH	D	DT	SH	N	PA1
Y	EH2	AH	UH2	ER	J	G	K	CH	NG	STOP
Y1	EH3	AH1	UH3	L	V		P	TH		
I	A	AH2	O	IU	THV			F		
I1	A1	AW	O1	U				H		
I2	A2	AW1	O2	U1						
I3	AY	AW2	OO	W						

TABLE 2  
Figure 9-13

Phoneme Categories According to Production Features  
 (courtesy of VOTRAX).

The largest category of phoneme sounds are voiced, but you also have voiced fricatives, voiced stops, fricatives, nasals, and three soundless phonemes available. Later on, you will see how the soundless phonemes are used to pace a phoneme string. In fact, putting pauses in the right places, even within a word, can make the difference between creating an understandable voice or gibberish.

When at all possible, you should refer to ready-made lists of words. VOTRAX for instance, has developed an extensive library of words with their phoneme breakdowns, examples of which are given in Appendix D. You can get additional assistance from a standard dictionary, where every word is phonetically represented, although each dictionary uses their own phonetic system. But, conversion charts for translating the dictionary symbols into sounds are usually included. Unfortunately, this type of matching procedure is rather difficult. You can, however, make your own phoneme strings for words through simple trial and error.

## Phoneme Strings

Thus far, you have been introduced only to the very fundamentals of phoneme determination. Now you must become acquainted with the techniques used for phoneme stringing and phoneme inflection enhancement, as well as the recognized technique for writing phoneme symbols.

Suppose, for example, that you want your synthesizer to say the word "sort." You could describe this word according to the ARPABET conversion discussed previously; in which case, you would get "s ow r t." This gives you a beginning, just as a regular dictionary would. However, don't try to convert words letter by letter. Instead, you should identify the number and type of sounds contained in the word. There will be at least one vowel, or one vowel combined with one or more consonant sounds in every English word. Speak the word aloud, carefully. Pay close attention to long or short sounds and pauses between sounds. Next, match each of the sounds you've identified with the appropriate symbol in the Phoneme Conversion Chart of Figure 9-12. Remember, the same procedure can be used to create any word in the English language.

As you begin to construct phoneme strings, you will recognize the need to add pauses or dead spaces at the beginning and end of some words. These pauses, or silent spots, are simply used to ensure that words in a phoneme string are not run together. Referring back to the Phoneme Conversion Chart of Figure 9-12, you can see that there are two pauses available to the user: /PA0/ which has a short time duration of only 47 ms and /PA1/ which has a longer time duration of 185ms. If an even longer pause is desired, you can lengthen it by simply repeating either of these phonemes.

Continuing with the phoneme string for “sort”, you will find that the fricative “S,” as used in the word “pass,” works well for the beginning of the first sound. This may be written as /S/. However, depending on how you pronounce the word “sort” you may feel it necessary to lengthen the “S” sound. You can do this by simply repeating it two or more times. If a pause were desired at the beginning of your phoneme string, you would begin writing your phoneme string as:

/PA1, S/ (for a short “S”)  
or  
/PA1, S, S/ (for a long “S”).

Continuing now with the “O” sound of “sort,” you may wish to use either the “O” phoneme from the word “cold” or the “AH” phoneme from “mop”. If you choose the “O” phoneme, notice that it has a much shorter time duration than the “AH,” therefore you may want to repeat /O/ in your phoneme chain. You may now finish this phoneme chain with the “R” phoneme from the word “red” and the “T” phoneme from the word “tap.” The final phoneme string, using the long “S,” the short “O,” and adding a pause at the beginning and end of the chain, would be written as follows:

/PA1, S, S, O, R, T, PA1/.

## Phonemes and Voice Inflection

The phoneme chain you have developed thus far is missing one important characteristic — that is voice inflection. If you were to program the word “sort” with only the basic phonemes /S, S, O, R, T/, the synthesizer would operate properly, however the output would be in a monotone. If all you were interested in was the synthesizing of a single word, a monotone voice may seem adequate. However, if your goal is a synthesizer that produces understandable phrases and sentences, you will have to program voice inflection, or pitch changes throughout the phoneme chains.

One of the advantages of the phoneme synthesizer over the stored-word synthesizer is its ability to change any word's inflection on command. Now consider how basic sentence meaning can be altered by voice inflection to produce statements, exclamations, or questions. As you can see, sentence inflection plays an important role in thought communication. Additionally, inflection changes are important for understanding of individual words. Especially, when those words are synthesized from a library of 40 or so common phonemes.

The word "sort," as you have already seen, is easy to reproduce using the phoneme chain /PA1, S, S, O, R, T, PA1/. As previously mentioned, these codes produce the basic phonemes, without inflection enhancement. Now look up the word "sort" in Appendix D. There, it is represented as follows:

2/S, 1/O, 1/R, 1/T.

NOTE: The phoneme dictionary does not include the pauses at the beginning and end of the words.

Note how the phoneme chain is written differently when variables other than the phoneme symbols are used in the string. Now let's break down VOTRAX's phoneme chain for "sort" to see how they added their inflection enhancement.

The numbers preceding the slantbars indicate each phoneme's inflection level, with larger numbers representing a higher basic pitch. Under the VOTRAX system, there are four possible inflection levels for any phoneme. Numberwise, these levels are represented as 0 — 3, with 0 implied with any phoneme symbols otherwise unmarked.

In "sort", our example word, each phoneme symbol has its own specified inflection level. When synthesized, the 2/S phoneme will have a higher basic pitch than the other phonemes. As you will see later, it is a relatively easy matter to insert inflection changes in a phoneme string.

## Phoneme Codes Versus Phonemes Symbols

You probably will not be able to insert phoneme symbols directly into most microprocessors. Instead, you will have to provide a translation between the standardized phoneme symbols and the microprocessor's I/O codes. This may mean translating the phoneme symbols into hexadecimal notation, as in the case of the ET-18 Robot Trainer. For example, the sample phoneme chain you developed for the word "sort" /S, S, O, R, T/ (shown without the pauses) would be rewritten into hexadecimal notation as 1F, 1F, 26, 2B, 2A. These hexadecimal equivalents for the phoneme symbols were taken directly from the Phoneme Chart of Figure 9-12, where they are listed under the heading "Hexadecimal Phoneme Code." As you may have noticed, there were no apparent provisions for including inflection levels with the phoneme codes. You will see how special software provisions can be developed for direct input of the four inflection levels later in this unit.



## Programmed Review

13.	The two primary methods of electronically producing synthesized speech are _____ and _____.
14.	(stored-words, stored phonemes) The stored-word method of producing synthesized speech is characterized by its ability to produce a/an _____ number of words. (limited/unlimited)
15.	(limited) A typical phoneme synthesizer chip operates at a data rate of _____ bits per second.
16.	(70) In a phoneme synthesizer circuit, the _____ acts much like the human vocal tract.
17.	(digital filter) When constructing phoneme strings it is necessary to include _____ or _____ at the beginning and end of the word.
18.	(pauses, dead spaces) The basic meaning of a word or sentence can be changed by changing voice _____.
19.	(inflection) The VOTRAX system of speech synthesis has _____ possible inflection levels.
	(four)

## USING A PHONEME SYNTHESIZER

The VOTRAX SC-01, a single-chip speech synthesizer, will provide you with an excellent example of the voice quality obtained by concatenating phonemes. This 22-pin large-scale-integrated circuit (LSI) contains all the circuitry necessary to generate phonetically synthesized speech. In this topic, we will show you how to interconnect, address, and operate the SC-01 phoneme synthesizer with a typical microprocessor.

### Interconnections

Figure 9-14 illustrates the type of connections required to make the SC-01 operate. Basically, you must provide the chip's supply voltage, timing, control, data addressing, and output audio amplification. Fortunately, the SC-01 chip demonstrates wide tolerances in regards to supply voltage, timing, and output audio circuits, allowing the chip to be used successfully in a wide range of equipment environments. Let's look at the more important SC-01 interfacing considerations.

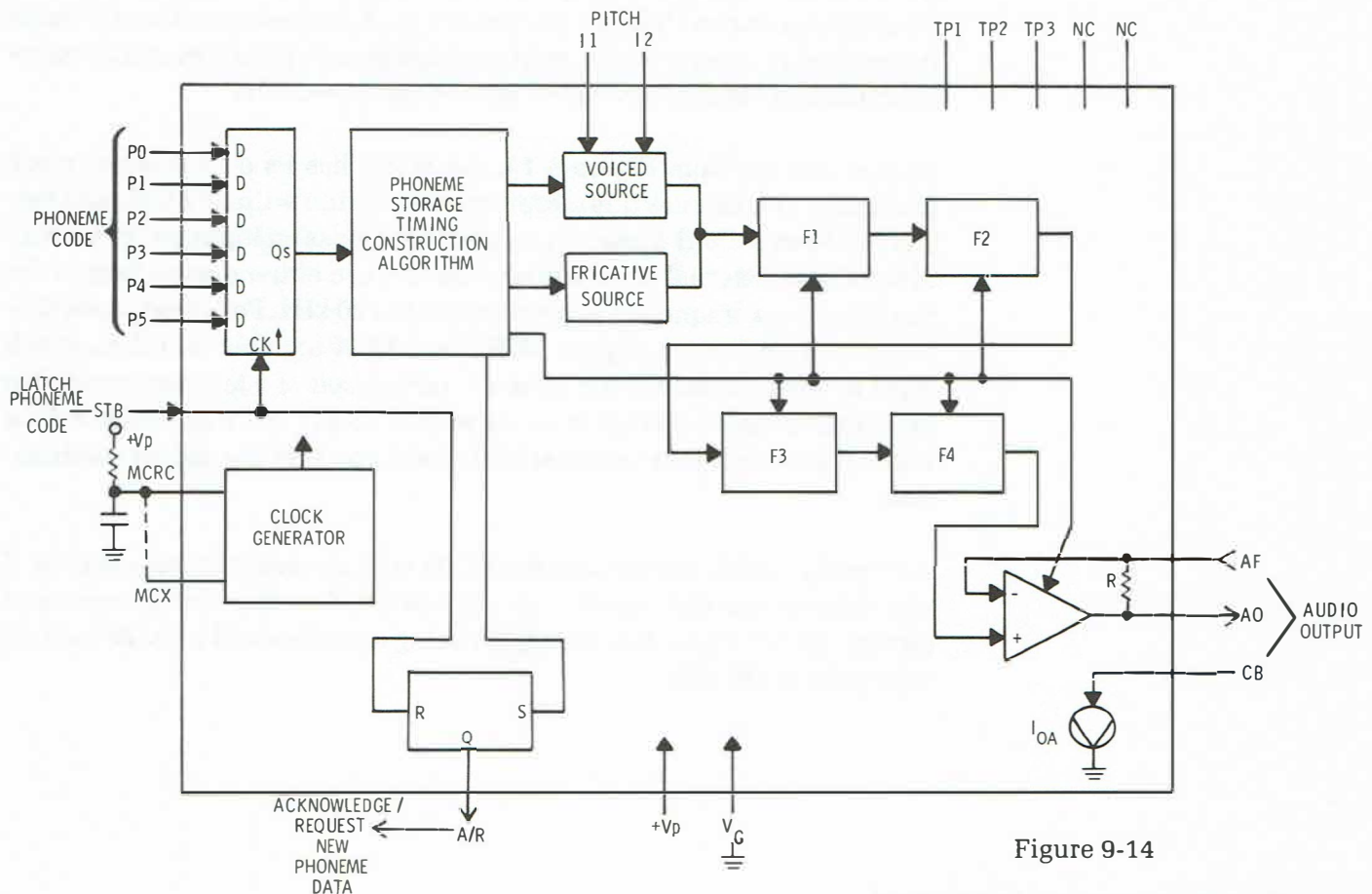


Figure 9-14

VOTRAX SC-01 phoneme synthesizer IC.

## SUPPLY VOLTAGE

The VOTRAX SC-01 is able to function within a supply voltage ( $V_p$ ) range of 7 to 14 volts. Normally, the chip's voltage reference ( $V_G$ ) pin, and any other pins requiring a permanent logic "0", are tied to the microprocessor's circuit ground. Then the chip's voltage source pin is tied to a stable positive voltage (nominally 12 volts) capable of providing 20 to 30 mA. Finally, the  $V_G$  of the chip should be coupled to circuit ground with a small value ceramic or tantalum capacitor.

## TIMING SOURCES

You can operate your VOTRAX SC-01 on its own internal timing generator or use an external clock frequency. The chip is internally optimized for a clock frequency of 720 kHz. Varying the clock frequency from this value changes the output speech frequency and phoneme timing. Even though you can use this technique to change the pitch of a particular phoneme, therefore altering the inflection in a word, you can get more consistent results by fixing the timing to the suggested frequency. Thereafter, you can change a phoneme's pitch frequency without altering the critical phoneme timing relationships by using the direct inflection/pitch control input lines, which will be discussed later.

As you can see from Figure 9-14, the SC-01 has its own internal clock generator. You can use this circuit in conjunction with an RC timing network to form a stand-alone timing generator, or as an input buffer for conditioning an external clock signal. You can use either case as long as the resultant clock frequency is approximately 720 kHz. For stand-alone timing, you must jumper inputs MCRC and MCX together and then attach them to a simple RC timing network, comprised of a temperature-stable capacitor of 300 to 330 pF in series with a 5.6 k $\Omega$  to 6.8 k $\Omega$  resistor. You will have to vary their values slightly until you find the proper combination.

As already stated, you can use the SC-01 with an outside timing source. If you wish to use this option, simply use MCX as the timing input and ground MCRC. Note that the input timing signal should provide switching levels at 720 kHz.

## CONTROL SIGNALS

There are three control signals associated with the SC-01's operation. The first and most important is the **strobe** (STB) signal. This signal, provided by the microprocessor, tells the SC-01 to accept the data levels present on the phoneme code inputs. The rising edge of this strobe signal is used to clock the data levels into the SC-01's input latches. The strobe signal also resets the Acknowledge/Request output, which is the second of the SC-01's control signals.

The **Acknowledge/Request** signal originates in the SC-01 and is used to inform the microprocessor of the SC-01's status. Once the strobe signal latches new phoneme data into the chip, the A/R signal is toggled to the zero state which inhibits the microprocessor from outputting the next strobe signal. The A/R signal returns to the one state after the SC-01 has had sufficient time to locate and begin phoneme construction. During this time, the next phoneme code is output from the microprocessor. Once the A/R signal goes high, the microprocessor outputs a strobe signal to the SC-01. Thus the Acknowledge/Request signal provides the necessary handshaking between the microprocessor and the phoneme synthesizer.

The third type of control signal originates from the microprocessor and takes the form of a two-bit binary **inflection code**. Altering this binary count instantly changes a phoneme's inflection level. As a result, you may program the synthesizer to produce four possible inflection levels for any of its 64 voiced phonemes. The two inflection level inputs (I1 and I2) alter the SC-01's output pitch. Figure 9-15 demonstrates the relationship of the binary inputs to phoneme pitch.

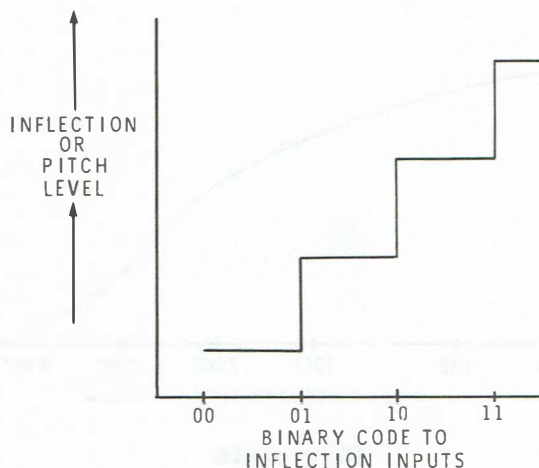


Figure 9-15

Phoneme inflection levels for VOTRAX SC-01 IC.

## PHONEME DATA

The phoneme data produced by the microprocessor is applied to the SC-01 as a six-bit parallel input. Actually, the SC-01 is designed to respond to an eight-bit data code, although only the first six bits are used to determine the phoneme choice. The remaining two bits are used to control the selected phoneme's pitch or inflection level. The six-bit binary phoneme code determines which of the synthesizer's 64 different phonemes are produced. Remember, the binary data on lines P0 through P5 is input to the SC-01's circuitry only during the rising edge of the strobe pulse.

## OUTPUT AMPLIFICATION

The VOTRAX SC-01 has its own built-in audio preamplifier. However, this circuit (which has only a low voltage, high impedance output) is inadequate for direct speaker applications. Therefore, you will have to provide an external audio amplifier; one that has sufficient power to drive a speaker. If you wish to obtain the most natural speech possible, the added amplifier should have its input frequency response tailored with a low pass filter. The purpose of the filter is to reduce the higher harmonics produced during fricative sounds. Figure 9-16 shows an example of a suggested amplifier audio response. Without the proper audio tailoring the SC-01's audio output becomes excessively harsh.

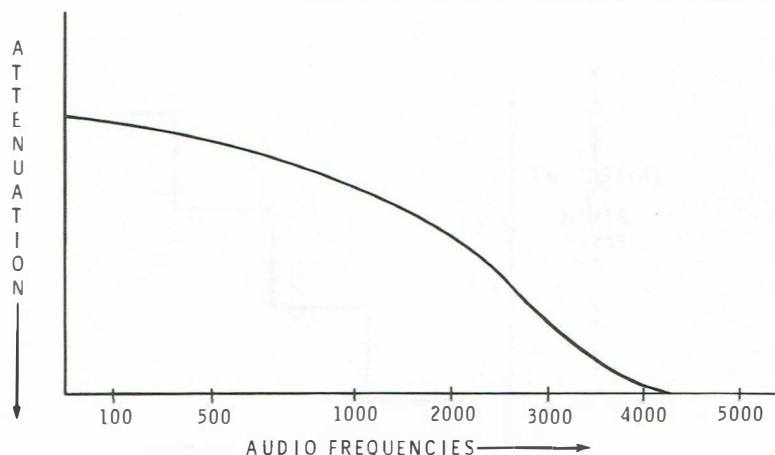


Figure 9-16

Typical frequency response for phoneme speech amplifier.



## Programmed Review

20.	The VOTRAX SC-01 is able to function within a supply voltage (Vp) range of _____ to _____ volts.
21.	(7, 14) The SC-01 _____ designed to operate from an external timing source. (is/is not)
22.	(is) For best results, the SC-01 should be operated at a frequency of approximately _____ kHz.
23.	(720) The _____ control signal, which originates in the SC-01, is used to inform the microprocessor of the SC-01's status.
24.	(Acknowledge/Request) The SC-01 has the ability to produce _____ basic phonemes.
25.	(64) You should provide _____ filtering to the synthesizer's output in order to reduce the higher harmonics produced during fricative sounds.
	(low pass)

## PROGRAMMING THE PSS

Thus far, you have learned how to develop a single word using the Phoneme Chart shown in Figure 9-12. You also saw how the Phoneme Dictionary, presented in Appendix D, became a valuable aid for quick development of a word. In the preceding topic, you observed how the VOTRAX SC-01 was connected to a typical microprocessor controller. You will now use this information to actually develop simple programs to make your Robot Trainer speak.

The ET-18 Robot Trainer is rather unique in that it uses two different methods by which to output speech. First, the Trainer has several, already developed, phrases stored in ROM. Second, the trainer has the capability of providing unlimited speech through user developed programs which are stored in RAM. We will discuss both of these methods starting with the preprogrammed, or "canned," phrases stored in ROM.

### Speaking With Canned Phrases

The following is a list of phrases, including their starting addresses, which have been preprogrammed into the ET-18 Robot Trainer's ROM.

<u>ADDRESS</u>	<u>PHRASE</u>
FA4B	Hello, my name is HERO.
FA64	Hello. I am HERO, The Heath educational robot.
FA93	I can talk, like this.
FAAA	I can move my arm.
FAC0	I can use my gripper.
FADA	I can turn my head.
FAF1	And I can move about.
FB09	Also, I can sense light, sound, and movement.
FB37	I see light.
FB45	Wow! It is dark.
FB56	Wait! Something moved.
FB6B	I heard that!
FB7B	There is something in my way.
FB9A	Help, help, help. Alarm, emergency.
FBC1	I have a brain, just like you do. But, my brain is a computer. My owner programs my computer for me and I always do as I'm programmed.
FC5B	There is no such thing as a bad robot, just a misprogrammed one.
FC9D	Gosh, I think I'm just about perfect.
FCC5	I think you are cute. Give me a hug, robots need love too.
FD04	You are very attractive for a human.
FD2C	Yes sir, you are handsome.
FD46	Your wish is my command.
FD60	Oh my! Please do not do anything to hurt me.
FD91	Please be quiet, I'm trying to sleep.
FDBA	I think I make an excellent pet, I'm even house trained.
FDF7	People stare at me a lot. I suppose it's because I'm so short.
FE33	Warning! Warning, intruder. I have summoned the police.
FE6D	Oh no! I do not do windows!
FE8C	I am incapable of making a mistake. Therefore, you must be wrong.
F2CC	Low voltage. (housekeeping phrase)
F4A1	Ready. (housekeeping phrase)

Since these phrases are programmed into the Trainer's ROM, there are two factors which must be taken into account. First, the ROM is "mask-programmed" by the manufacturer; therefore, the contents of the ROM cannot be altered in any way. Thus, the user is not able to change phrase wording, inflection, or add and delete pauses. However, the second factor is of great benefit to the user. Because the phrases are programmed into ROM, they will remain there at all times, even when power is turned off. A more detailed discussion of ROMs will be presented in the next unit.

Even though you are not able to change the canned phrases, they do have several advantages. They are easily accessible to the user through keyboard entry. For instance, the Trainer can be programmed to speak the phrase "There is something in my way," located at ROM address FB7B, using the following simple keyboard entries:

<u>KEY STROKE</u>	<u>COMMENT</u>
RESET	Enter the executive mode.
A	Enter the repeat mode.
A	Use automatic memory address increment.
0100*	Memory address for program beginning.
72	Specifies the Trainer Speech Mode.
FB7B	Specifies address of phrase to be spoken.
20FE	Stops further execution of instructions.
RESET	Reenter executive mode.

The Trainer will now speak the phrase when you enter the following commands:

<u>KEY STROKE</u>	<u>COMMAND</u>
A	Enter the repeat mode.
D	Do the program starting at address.....
0100	The address where the program starts.

In the preceding illustration, you simply inserted the robot speak command (72), followed by the address where you wanted the program to start, followed by the ROM address of the phrase you wanted the robot to speak. When you run the program, the robot will enter the speech mode, go to the beginning address, speak the phrase, exit the speech mode, and return to the next step in the program. As you can see, it would be very easy to enter a canned phrase anywhere in a programmed routine.

\* This can be any usable address where you want the program to begin.

It is also very easy to combine two phrases to make the robot speak a longer phrase. For example, if you desired to have the robot combine the phrase "I heard that," located at address FB6B, and the phrase "Please be quiet, I'm trying to sleep," located at address FD91, you would simply enter the program as follows:

<u>KEY STROKE</u>	<u>COMMENT</u>
RESET	Enter the executive mode.
A	Enter the repeat mode.
A	Use automatic memory address increment.
0100	Memory address for program beginning.
72	Specifies the speech mode.
FB6B	Specifies address of first phrase to be spoken.
72	Specifies the speech mode.
FD91	Specifies address of the second phrase to be spoken.
20FE	Stops further execution of instructions.
RESET	Reenter executive mode.

As shown, the speak command (72) has to be entered twice for this program; once to speak the first phrase, and again to speak the second phrase. This is necessary since the last few commands of each canned phrase tells the microprocessor to exit the speech mode and return to the next step in the program. If you did not enter the second speech command (72), the microprocessor would not know what to do next. Thus, it would just wait at the end of the first phrase.

The program is executed in the same manner as the previous single-phrase program.

If desired, you could string all the phrases together, in any sequence you choose, by simply adding a speech command (72) between the phrases. You could also include canned phrases anywhere in a given routine by simply entering the speech command followed by the appropriate phrase.

As you can see, the canned phrases are very simple to use and are somewhat similar to the stored-word method of providing speech. Remember, when you use canned phrases, you do not have to bother with programming specific words, adding inflection levels, or entering pauses. However, you do not have control over them either. Let's now take a look at how you can give the robot an unlimited vocabulary.



## Programming Unlimited Speech

Programming the Trainer to speak user developed words or phrases is quite simple. However, it is a rather time consuming process when complete words or phrases must be developed from scratch. In the first portion of this topic, you will learn how to develop and program single words. Later, you will see how complete phrases or sentences are developed and programmed.

### PROGRAMMING SINGLE WORDS

In our first example, we will develop a simple program to speak the word "transfer." The first step in the process is to develop the word you wish to program. However, before you start to develop the word from scratch, using the Phoneme Chart shown in Figure 9-12, you should check the Phoneme Dictionary in Appendix D or the dictionary supplied with the Robot's speech accessory to see if the word has already been developed. Checking Appendix D, you will find that the word "transfer" has already been developed. The word appears as:

TRANSFER      2/T 2/R .... 1/AE1 2/I3 1/N 2/S 1/F 1/ER  
AA, AB, 6F, 89, 4D, 9F, 5D, 7A

The first line 2/T 2/R .... 1/ER shows each phoneme symbol required to develop the word. Recall, the number before the / designates the inflection level given to that particular phoneme. Inflection levels will be discussed in greater detail when we form a word from scratch. The second line AA, AB, .... 6F shows the hexadecimal equivalent for each phoneme used to develop the word. These hexadecimal equivalents are the data used to program the Robot to speak.

Before we write the program, one important factor must be considered. The program must have the ability to store the complete sequence of phonemes required to speak the word "transfer" and all the commands required to control the Robot's speech. The most straightforward approach for storing this data is to construct a temporary storage buffer in the microprocessor's random access memory (RAM). Using this method, you are theoretically unlimited in the total number of phonemes you may chain together. However, for practical purposes, you must confine a key-stroke buffer to a size consistent with the usable memory of the Robot Trainer.

Now that you understand the necessity of including a key-stroke buffer into your program, let's see how relatively simple it is to write a routine which will store the word "transfer" in memory. The program would be as follows:

KEY STROKE	COMMENT
RESET	Enter the executive mode.
A	Enter the repeat mode.
A	Use automatic memory address increment.
0100	Memory address for program beginning.
72	Specifies Trainer speech mode.
0105	Tells where the actual phoneme information for the word "transfer" starts (actually entered two digits at a time — "01" "05") since 0100 through 0104 have 72, 01, 05, 20, FE in them.
20FE	Tells the microprocessor to wait here after outputting speech.
AA	Hexadecimal code for the phoneme symbol T, (raised 2 inflection levels).
AB	Hexadecimal code for the phoneme symbol R, (raised two inflection levels).
6F	Hexadecimal code for the phoneme symbol A1.
89	Hexadecimal code for the phoneme symbol I3, (raised two inflection levels).
4D	Hexadecimal code for the phoneme symbol N, (raised one inflection level).
9F	Hexadecimal code for the phoneme symbol S, (raised two inflection levels).
5D	Hexadecimal code for the phoneme symbol F, (raised one inflection level).
7A	Hexadecimal code for the phoneme symbol ER, (raised one inflection level).
3F	STOP command.
FF	End of speech.
RESET	Return to executive mode.

#### PROGRAM NOTES:

1. The program does not need to start at address 0100. You can start at any memory address of 0050 or higher. However, the end of the program cannot extend beyond the address 0ED0. Memory addresses below 0050 and above 0ED0 have been reserved for the Robot's monitor program.
2. You can store a number of speeches at different addresses, and call up the one you want by sending the computer to the proper address.

3. You need an 03 (short silent pause), 3E (long silent pause), or 3F (STOP) at the end of each speech routine, or the voice will trail off with a funny sound.
4. If you turn the Robot off, it will forget the speech program and you will have to reenter it.

You execute the program, to make the Robot speak the word “transfer,” in exactly the same manner as you did to make it speak canned phrases.

<u>KEY STROKE</u>	<u>COMMENT</u>
A	Enter repeat mode.
D	Do the program starting at address.
0100	The address where program starts.

To program the Robot to speak a word that isn’t included in the Phoneme Dictionary is much more time consuming, but rewarding. Before you program your Robot to speak the desired word, you must first develop the word. To develop a word from scratch, it is necessary to become thoroughly familiar with the Phoneme Chart of Figure 9-12. Until then, you will basically be using the trial-and-error method to develop your words.

Before you try and develop and program a specific word from scratch, carefully study the helpful hints listed below.

### **Helpful Hints**

1. Different phonemes may produce the same sound, but different time durations (length of sounding). Listen to “EH3,” “EH2,” and “EH1,” then listen to “I3,” “I2,” and “I1.” The difference in their length may be what you need to fix a word that sounds “clipped” or “lopsided.”
2. Remember that you are free to use two or three of the same (or similar) phonemes in a row, to smooth out a word’s sound. Usually, you will use vowel sounds (A, E, I, O, U) for this.
3. Cultivate a good ear for speech patterns and word sounds. For instance, the word “I” is a diphthong (that is, it contains two sounds, “AH” and “E”). With work, you will recognize special sounds, accents, and changes of pitch (which you will be able to reproduce using the inflection capability).

4. When you first program "by ear," insert extra silent pauses between the phonemes. This will give you space to insert phonemes, if needed, when you adjust your program for better sound; by providing the extra spaces, you can add a phoneme without having to reenter the rest of the program. In addition, it will be easier to hear the separate phonemes, so you can analyze and adjust them.
5. Become very familiar with the process for changing entries in a program, since you are sure to want to modify the sounds after you hear them a few times. Once a program is in memory, you can review it and make any changes you need using the following procedure:
  - (A) Press the "A" key.
  - (B) Press the "E" (EXAM) key to examine the memory.
  - (C) Enter the memory address where you wish to start examining.
  - (D) Press the "F" (FWD) and "B" (BACK) keys, so you can look forward and backward through the memory addresses, until you have seen all the address you wish to examine. The left four spaces on the display will show the memory address, and the right two spaces will show the command or data that is contained in that address.
  - (E) If you see information in a memory address that you wish to change, press the "C" (CHAN) key. The right two spaces in the display will go blank, allowing you to enter the new information you wish to place there.
  - (F) You may continue to go forward and backward, and change information until you are satisfied with the word. Then simply exit the examine operation by pressing RESET.
6. Be sure to write down the phoneme codes for words that seem to work especially well for you. Choose a good place for saving these homemade words, such as the blank pages in the back of your Robot dictionary.

Let's now develop the word "kingdom." Once the word has been developed on paper, we will construct a simple program to make the Robot speak the word. The first step in word development is to look up the desired word in a good dictionary and, write the phonetic equivalent of the word on a piece of paper. The phonetic equivalent of the word "kingdom" is:

\`k i ŋ — d ə m\  
 — — — — —

As shown, there are six distinct sounds in the phonetic spelling of the word kingdom. You may have to spend a few minutes studying the phonetic symbols in your particular dictionary in order to understand their meanings. To help you develop the word, draw a line below each of the phonetic sounds in the word, as shown above.



Next, refer to the Phoneme Chart of Figure 9-12 and find the phoneme symbol that best represents the “k” sound in the word kingdom. In this case, there is only one phoneme symbol for the “k” sound, that of the word “trick.” Now, enter the hexadecimal phoneme code, in this case 19, which represents the phoneme symbol for the “k” sound in the space below the “k.” Your phonetic spelling should now look like this:

```
\`k  i  ŋ  —  d  ə  m\
  19
—  —  —  —  —  —
```

Moving to the “i” sound in the phonetic spelling of the word kingdom and referring to the Phoneme Codes of Figure 9-12, we find that there are three possible phoneme symbols that can be used to represent an “i” sound — I3, I2, and I1. Here is where the trial-and-error portion of word development comes into play, until you become familiar with the various sounds of the phoneme codes. If you decided upon the I1 phoneme symbol, you would write the hexadecimal phoneme code (0B) in the space under the “i.” Your phonetic spelling would now be:

```
\`k  i  ŋ  —  d  ə  m\
  19 0B
—  —  —  —  —  —
```

The next phonetic sound is the “ng” sound as in the word “thing.” Thus, entering the appropriate hexadecimal phoneme code, our phonetic word now looks like this:

```
\`k  i  ŋ  —  d  ə  m\
  19 0B 14
—  —  —  —  —  —
```

You would develop the rest of the word in the same manner, putting the appropriate hexadecimal phoneme code in the space under the corresponding phonetic sound of the word. Your developed word could look like this,

```
\`k  i  ŋ  —  d  ə  m\
  19 0B 14  1E 32 0C
—  —  —  —  —  —
```

depending on your preference of sounds and accents you are familiar with hearing.



You can now use the hexadecimal phoneme codes you just developed, which correspond to the phonetic spelling of the word “kingdom,” to make your Robot speak. Your program is written, entered through the keyboard, and executed in exactly the same manner as previously done for the word transfer, which you found in the Phoneme Dictionary. The program would be entered as follows:

ADDRESS	KEY STROKE	COMMENT
*	RESET	Enter the executive mode.
*	A	Enter the repeat mode.
*	A	Use automatic address increment.
*	0100	Memory address for program beginning.
0100	72	Specifies Trainer speech mode.
0101	01	Tells where the actual phoneme information for the word “kingdom” starts.
0102	05	Tells the microprocessor to wait
0103	20	here after outputting speech.
0104	FE	Short pause.
0105	03	Hexadecimal code for phoneme K.
0106	19	Short pause.
0107	03	Hexadecimal code for phoneme I1.
0108	0B	Short pause.
0109	03	Hexadecimal code for phoneme NG.
010A	14	Short pause.
010B	03	Hexadecimal code for phoneme D.
010C	1E	Short pause.
010D	03	Hexadecimal code for phoneme UH1.
010E	32	Short pause.
010F	03	Hexadecimal code for phoneme M.
0110	0C	STOP command.
0111	3F	End of speech.
0112	FF	Return to executive mode.
*	RESET	

You could now execute the program and check the quality of the word. If you wanted to change any of the phonetic representations, you would just follow the procedure outlined in the helpful hints.

Once you have obtained the quality of the word you desire, you simply re-write and enter the program omitting the pauses. Your new program would then look like this:

<u>ADDRESS</u>	<u>KEY STROKE</u>	<u>COMMENT</u>
*	RESET	Enter the executive mode.
*	A	Enter the repeat mode.
*	A	Use automatic address increment.
*	0100	Memory address for program beginning.
0100	72	Specifies trainer speech mode.
0101	01	Tells where the actual phoneme
0102	05	information for the word "kingdom" starts.
0103	20	Tells the microprocessor to wait
0104	FE	here after outputting speech.
0105	19	Hexadecimal code for phoneme K.
0106	0B	Hexadecimal code for phoneme I1.
0107	14	Hexadecimal code for phoneme NG.
0108	1E	Hexadecimal code for phoneme D.
0109	32	Hexadecimal code for phoneme UH1.
010A	0C	Hexadecimal code for phoneme M.
010B	3F	STOP command.
010C	FF	End of speech.
*	RESET	Return to executive mode.

## Inflection Enhancement

Without inflection enhancement, even a perfectly constructed phoneme string sounds dull and droning, if not misleading in context. Perhaps the most obvious use of inflection is to make a question, an exclamation, or a simple statement from the same word or phrase. A question is usually identified by an ending rise in pitch, although this ending rise in pitch is sometimes followed by a very weak drop. In either case, the word tends to be somewhat pitched and the change in tone is a sliding or gradual one. Questioning inflection can be difficult to do with a phoneme synthesizer that has only four distinct inflection levels. Improper use of phoneme stringing and inflection changes often result in chopped or stepped words or phrases, which seldom sound like questions. However, with proper precautions, you can avoid such problems.

You will recall that the pitch inputs for the VOTRAX SC-01 are independent from the phoneme inputs. Thus, you are theoretically able to modify any single phoneme's inflection a number of times. Even though the SC-01 has built-in circuitry that compares phonemes to assure smooth transitions from one to the next, **the chip can not compensate for inflection changes that occur at times other than at phoneme switching.**<sup>†</sup>

<sup>†</sup>With special programming techniques, inflection levels can be changed in mid-phoneme if desired. These special programming techniques will not be discussed since they are beyond the scope of this course.

Therefore, any software program designed to permit changes of inflection should also be designed to only output those changes at phoneme transition time. If you follow this precaution, you are less likely to accidentally generate chopped or stepped words and phrases.

Another reason for inflection changes is simply to make words more natural, or human sounding. However, you should use moderation when adding inflection changes to a word, as rapid or extreme changes could easily cause a “sing-song” effect, which is little better than the original monotone as far as naturalness is concerned.

The words included in your Phoneme Dictionary already have the appropriate inflection levels included in the hexadecimal phoneme codes. However, when developing words from scratch, a good standard dictionary is again a valuable aid. The dictionary will show where, and how much, emphasis is placed on certain sounds of specific words. The dictionary will also show if certain sounds are long or short; if they sound like another letter or sound; and whether the main emphasis is placed at the beginning, middle, or end of the word. It is recommended that you become thoroughly familiar with your particular dictionary before you attempt to add inflection to your developed words. This familiarization will save you time and frustration when you develop words that are not included in the Phoneme Dictionary.

Recall, the first six bits of the 8-bit phoneme/inflection byte are used to enter the hexadecimal phoneme code. You will now see how the last two bits, the most significant bits (MSB's), are used to enter the inflection level. The Phoneme Chart of Figure 9-12 shows all hexadecimal phoneme codes for one set of sounds, those spoken at the lowest pitch. For example, the hexadecimal phoneme code (19) was used to produce the “k” sound in developing the word kingdom. The hexadecimal phoneme code (19) represents the lowest possible pitch for the “k” sound. Thus, the 8-bit phoneme/inflection byte representing the lowest pitch for the “k” sound is 00011001, which is hexadecimal 19.

You can program sounds at the next higher pitch, or inflection level, by simply adding the number “hexadecimal 40” to any hexadecimal phoneme code. Therefore, if the next higher pitched “k” sound is desired, you simply add hexadecimal 40 to the hexadecimal phoneme code 19. Now, the 8-bit phoneme/inflection byte representing the “k” sound, raised one inflection level, is 01011001, or hexadecimal 59, which is hexadecimal 40 added to the original hexadecimal 19.

Similarly, you can raise the pitch level again by adding another hex 40, and raise it even again by adding yet a third hex 40. As shown in the example below, there are four inflection levels for any hexadecimal phoneme code.

<u>BASIC PHONEME CODE</u>	<u>RAISED ONE LEVEL</u>	<u>RAISED TWO LEVELS</u>	<u>RAISED THREE LEVELS</u>	<u>SOUND PRODUCED</u>
00	40	80	C0	“EH3”
19	59	99	D9	“K”
2E	6E	AE	EE	“AE”
3D	7D	BD	FD	“AW”

NOTE: The lowest pitch, both inflection bits low, produces the longest duration for a given phoneme sound.

## Programmed Review

26.	The contents of the canned phrases used in the ET-18 Robot Trainer _____ be changed. (can/cannot)
27.	(cannot) When you program the ET-18 Robot Trainer to speak more than one canned phrase during a routine, you must enter the _____ command (72) before you enter the address of the next canned phrase to be spoken.
28.	(speak) Once a word has been developed and programmed into the Trainer, it _____ possible to change the contents of the word. (is/is not)
29.	(is) Referring to the Phoneme Chart shown in Figure 9-12, a hexadecimal phoneme code of A6 would represent the phoneme symbol "O" raised to the _____ inflection level.
(second)	



## NATURALNESS AND PHONEME PHRASES

Until now, your phoneme strings were limited to single words. As you well know, few ideas can be effectively communicated in that manner. Therefore, you must be able to create phoneme string equivalents for required phrases or sentences. Additionally, you must build your strings so that they will communicate effectively. To accomplish these tasks, you not only use those fundamentals of phoneme concatenation, but you must also learn to incorporate additional inflection and timing modifications. If you attempt to construct your phoneme phrases by literally lifting their values directly from the Phoneme Dictionary, you will undoubtedly be very disappointed with the end results.

Part of that natural quality of the human voice which tends to be missing in the phoneme synthesizer's output is typically an intonation or inflection problem rather than an incorrect choice of phonemes. How do you know what inflection levels to assign each phoneme in a string? To answer this question, you must take a more in-depth look at the entire phoneme speech concatenation process.

## The Phoneme Concatenation Process

The entire phoneme concatenation process involves three major steps. To begin with, you must determine both the basic contents of the entire phoneme string, and its primary communicative task. In the second step, you determine your options. In other words, how you could modify the phoneme string to change and, therefore, improve its overall clarity. In the third and final step, you utilize your findings from step two, to modify the original phoneme string.

### THE FIRST STEP — THE BASIC PHONEME STRING

Before attempting any phoneme modifications, you first need to develop your basic phoneme string. By now, you should find it relatively easy to determine the essential phonemes for most phoneme phrases. Let's study the following example as a starting point for our discussion:

“These are your options”

The quickest means of building any basic phoneme string is to first check your Phoneme Dictionary in Appendix D or the dictionary supplied with the Robot's speech accessory. If a particular word you seek is not in the dictionary's listing, you will often find a combination of words that include the same sound patterns. As a last resort, you can always refer to the phoneme tables to build your word from scratch.

Whether you copy your phoneme strings from the dictionary, or build them yourself, your first phrases will normally sound like a meaningless collection of words. This is usually due to an absence of any verbal punctuation in the phoneme phrase. After all, the phoneme chip can only produce sounds as they are programmed. Therefore, if your phoneme phrases are not properly modified, your synthesizer's output will tend to be a simple collection of independent words, rather than a unified thought or idea.

Take another look at the following sample phrase:

“These are your options”

When you read a nonpunctuated phrase like this, you are unable to determine whether it was intended to present a question, or an exclamation, or perhaps a simple statement of fact. That does not mean to say that a phoneme phrase built from such a collection of words is useless. In fact, you will normally build your phoneme phrases on a straight sound-by-sound basis, temporarily ignoring the effect of punctuation marks.

So the first step in stringing phoneme phrases is to list only the basic phoneme sounds of each word in the phrase, thus ignoring any possible pronunciation changes due to punctuation. Therefore, you could begin building a phoneme string for the example phrase, even though there are no punctuation marks shown.

Look at the following phoneme listing:

/PA1, THV, E1, E1, Z, PA1, AH1, UH3, ER, PA1, Y, O, R,  
PA1, AH1, P, SH, UH3, N, Z, PA1/

This very basic phoneme equivalent to the example phrase is technically accurate, but only on a sound-by-sound basis. Consequently, you would be quite dissatisfied if you programmed it directly to your synthesizer. In this case, the resultant speech would be just as bland and meaningless as the original unpunctuated phrase.

The next step in preparing your phoneme phrase is to include the more recognizable pronunciation changes caused by punctuation marks. The most noticeable of these changes are:

1. Extended pauses between individual thoughts.
2. Sound emphasis for exclamations.
3. Sound pitch fluctuations characteristic of sentence types.

In the following sections, you will see different techniques used to implement these pronunciation changes in phoneme phrases. You will also see how punctuation marks are not the only means of manipulating the meaning of a group of words, especially in verbal communication.

## THE SECOND STEP — KNOW YOUR OPTIONS

One of the more interesting features of the English language is the inherent ability of the speaker to produce subtle, but important changes in a word's meaning through sound intonation. Intonation is probably the most noticeable method used to modify word sounds, although there are other more subtle changes, such as emphasizing pauses, which are not as clearly defined.

Once you have a basic listing of phonemes, you need to ascertain the purpose of the original word phrase. Was it intended to ask a question, state a fact, or perhaps voice a warning? Of course there are many different purposes, which we will call tasks, for a phrase or sentence, other than the three just mentioned. However, they are not usually as easy to define, let alone synthesize.

### Pauses

As you begin to modify your first phoneme phrases, you will undoubtedly find numerous opportunities to add subtle, yet meaningful pauses between certain words. Many of these pauses are dictated by punctuation marks. Others are inserted purely as a matter of style or personal emphasis. For example, you can place emphasis on the word "your" by increasing its separation from the words immediately before and after.

Another technique used by all of us to emphasize certain words is through sound volume. Unfortunately, the typical phoneme circuit has a fixed output level, therefore making it impossible to add this type of emphasis. Fortunately, most people provide voice emphasis more by the use of timing pauses.

### Intonation

Timing pauses are sometimes more a matter of taste or preference, therefore becoming a nice-to-have option, but proper intonation techniques are essential to the understanding of the word phrase's task.

Once you have identified the important sound pauses in your word phrase, you study its intonation. If you are not totally familiar with this term, remember that “intonation” refers to the pattern of pitch changes in speech. To understand this point a little more clearly, study the sample intonation graphs in Figure 9-17. The bar line graph below each sentence is used to reflect that word group’s intonation. You will notice that the differences depend primarily on whether the task is to present a statement, an exclamation or a question.

The graphs in Figure 9-17 do not represent the only means of intoning a statement, exclamation, or question. You may often use different combinations of sound patterns for the same tasks; however, these are the most common. The object of the intonation bar graph is simply to plot the most significant pitch changes throughout the entire word group. After you have this graph plotted, you can then modify your phoneme phrases to match.

When you are dealing with phoneme strings, you may not always be able to stop at intoning the phrase in general. Instead, you will often find it necessary to perform similar operations with each individual word in the group. This extra graphing gives you an opportunity to foresee possible pronunciation problems in the individual phoneme word chains. Remember that the phoneme synthesizer utilizes a fixed assortment of phonemes; and that these phonemes, when unmodified, are all at the same approximate pitch.

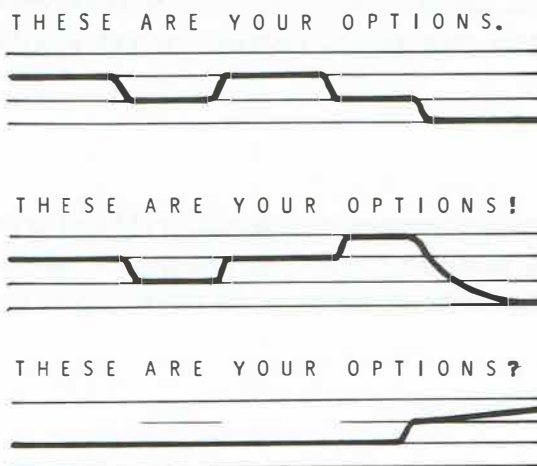


Figure 9-17

Typical intonation bar graphs for different sentence tasks.



The example in Figure 9-18 shows how dual intonation bar graphs are used. Study this example closely. You should notice the following:

1. The task graph is far smoother in its transitions than the word graph.
2. There are four levels of intonation indicated for both graphs.
3. That both graphs have their average intonation levels in the middle range.

The task graph should always indicate the major trend of the entire word group. It should not try to reflect those intonations necessary for any single word's understanding. However, this graph will often reflect a major intonation in key words, such as the last word in a question.

The number of indicating levels for both the task and word intonation graphs depends on the type of phoneme synthesizer used. Those in this course are set to represent four distinct levels, which happens to reflect the total range of inflection changes allowed by the VOTRAX SC-01. Naturally, you would use whatever scale is appropriate to your own phoneme synthesizer.

The last point to mention on Figure 9-18 is that the average level of both task and word intonation is normally set to the mid range. This is the only way that you can be assured the ability to reflect both higher and lower intonations.

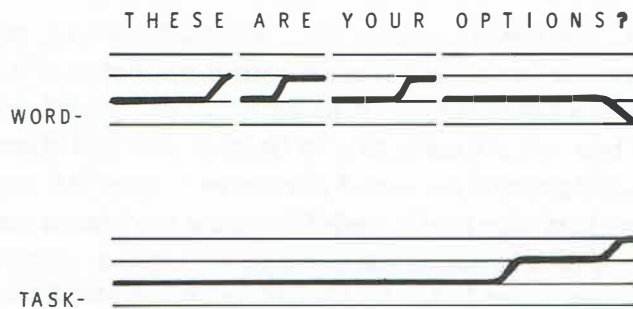


Figure 9-18

Intonation bar graphs for the individual words as well as the main sentence tasks.

### STEP THREE — MODIFYING TIMING AND INTONATION

You now have the essential background information required for both writing and modifying phoneme phrases or sentences. All that remains is for you to see a few examples of how the entire process is performed.

#### Adding Word Emphasis

You may use two basic methods to emphasize any particular word in your phoneme phrases. The first is to lengthen its basic sound duration. The second, and the preferred method, is to provide an increased isolation between the emphasized word and those surrounding it. In other words, you increase the pauses.

In the phrase, “these are your options”, you could put additional emphasis or impact on any one word by either stretching its sound or by lengthening certain pauses. For instance, you could emphasize the word “your”, /PA1, Y, O, R, PA1/, by increasing the sound duration of one or more of its phonemes. Better yet, you could isolate it from the words “are” and “options” by increasing the length of the separating /PA1/ phonemes. You may even wish to use a combination of both methods. The following examples and their descriptions in the next paragraph should help illustrate these methods:

1. /PA1, Y, O, O, R, PA1/
2. /PA1, PA1, Y, O, R, PA1, PA1/
3. /PA1, Y, O, R, PA1, PA1, PA1/
4. /PA1, PA1, Y, O, O, R, PA1, PA1/

In example 1, the vowel sound “ou” is lengthened by using the /O/ phoneme twice. In example 2, the no-sound phoneme, /PA1/, is duplicated to provide a slight timing isolation between “your” and the rest of the phrase. When you are using this technique, you will often have to use more than two additional no-sound phonemes to gain the required pause effect. And you may also find it desirable to use uneven pauses, as shown in example 3. A final possibility is given in example 4, where a combination of techniques is used. As you should see, word emphasis, regardless of the method, is basically a trial-and-error process.

### **Word De-emphasis**

It is extremely difficult to de-emphasize a word in a phoneme string. For example, you saw how easy it was to emphasize the word “your” in the sample phrase. You simply had to either lengthen the “ou” sound, increase the pause lengths, or use a combination of both. Unfortunately, the opposite rules do not hold true for word de-emphasis. First of all, there are few phoneme sounds which can be shortened in length. Especially if you don’t wish to change other important characteristics, such as their basic pitch frequency. And secondly, any attempt by you to shorten the no-sound phonemes will normally result in slurred speech. Therefore, neither of the techniques used to emphasize a word in your phoneme strings can be successfully modified to de-emphasize one. You may, however, find it possible to de-emphasize some words by carefully emphasizing those around it.

### **Modifying Intonation**

The next step in modifying your phoneme phrase is to add the appropriate inflection characteristics. But in order to accomplish this type of modification, you must first take into account the type of phoneme synthesizer used.

A typical phoneme synthesizer will allow you to modify your phoneme’s basic pitch. This modification occurs through programmable changes in the synthesizer’s voiced source timing. In effect, you can cause the synthesizer to speed up or slow down its sound source generator by manipulating the logic levels on its two inflection input pins (I1 and I2).

When it is programmed properly, a phoneme synthesizer is able to provide fairly smooth-sounding audio, regardless of the timing shifts caused by changing inflection codes. Unfortunately, the most difficult part of programming intonation into your phoneme strings is determining where to make each inflection level change. You can greatly reduce the difficulty of these determinations, and improve the resultant audio quality, by using the following guidelines:

1. Avoid changing inflection levels during a phoneme.
2. Avoid jumps of more than one inflection level per phoneme in any word.
3. Keep your average phoneme inflection level in the midrange of the synthesizer's capabilities.
4. Avoid changing inflection levels between two identical phonemes.
5. Program the inflection of important words first. Then modify the entire string for task intonation.

Remember, just as it is nearly impossible for two individuals to speak a sentence identically, it is more so for you to attempt programming your phoneme strings to match every subtle change in your own voice. However, you still need to identify important intonations and then modify your phoneme strings accordingly. Only in this way, can you truly obtain the most natural synthesized speech.

## Programmed Review

30.	The phoneme concatenation process involves _____ major steps.
31.	(three) The quickest means of building any basic phoneme string is to first check your _____ dictionary.
32.	(phoneme) Intonation is probably the _____ (most/least) noticeable method used to modify word sounds.
33.	(most) When using the SC-01 voice synthesis chip, emphasis _____ be accomplished by changing the vol- (can/cannot) ume of the word or sentence.
34.	(cannot) You may emphasize a word by either _____ (shortening/lengthening) its overall sound or by _____ the separating (increasing/decreasing) pause durations.
35.	(lengthening, increasing) The intonation _____ is a convenient method of plotting the most significant pitch changes throughout an entire word group.
36.	(bar graph) By speeding up or slowing down the synthesizer's _____ you can, in ef- fect, modify your phoneme's basic pitch.
	(sound source generator)



## EXPERIMENT

Perform Experiment 15 in Unit 12. Once you have completed the experiment, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is the most correct. When you have completed the examination, compare your answers with the correct ones that appear after the exam.

1. A language's smallest fundamental unit is:
  - A. A phone.
  - B. An allophone.
  - C. A phoneme.
  - D. A word.
2. Which of the following devices is used to electronically examine the speech waveform?
  - A. The holograph.
  - B. The orthograph.
  - C. The spectrograph.
  - D. The pantograph.
3. Which of the following sounds is an example of a sibilant?
  - A. /f/.
  - B. /p/.
  - C. /n/.
  - D. /s/.
4. The sound created by closing off the air stream completely is called:
  - A. Voiced.
  - B. A fricative.
  - C. Noiseless.
  - D. A stop.
5. The term articulation refers to:
  - A. The process of slowing down speech.
  - B. The process of shaping speech using the tongue, teeth, and lips.
  - C. The process of writing phonemic transcriptions.
  - D. The process of softening vowels during speech.

6. Physically lengthening the pharynx:
  - A. Raises its resonant frequency.
  - B. Lowers its resonant frequency.
  - C. Exercises the velum.
  - D. Is impossible.
7. The basic phoneme synthesizer is able to produce:
  - A. An unlimited selection of words or sentences.
  - B. A limited selection of words or sentences from voice imprints stored in an external read-only-memory.
  - C. High quality words using waveform-encoding techniques.
  - D. A specific vocabulary as found in its internal read-only-memory.
8. The phoneme parameters found in the synthesizer's look-up table determine:
  - A. The digital code to be output to the processor.
  - B. The duration of the phoneme produced.
  - C. The loudness of any particular word.
  - D. The number of phonemes stored in read-only-memory.
9. The typical phoneme synthesizer operates:
  - A. Independent of any external data processing.
  - B. At a data rate of 1200 to 2000 bits per second.
  - C. At a data rate of approximately 70 bits per second.
  - D. At a fixed level of output tone or frequency.
10. Which of the following represents the largest category of phoneme sounds?
  - A. Voiced.
  - B. Voiced fricatives.
  - C. Voiced stops.
  - D. Nasals.

11. Which of the following is not a required input to the VOTRAX SC-01?
  - A. An external timing signal at 720 kHz.
  - B. A 6-bit phoneme data code.
  - C. A strobe signal.
  - D. A power source between 7 and 14 volts.
12. In order to change the inflection of a phoneme, you would:
  - A. Input a different 6-bit phoneme code.
  - B. Change the frequency of the strobe signal (STB).
  - C. Change the 2 most significant bits of the 8-bit phoneme code.
  - D. Change the level on the Acknowledge/Request (A/R) line.
13. Which of the following is NOT determined by the phoneme synthesizer's support software?
  - A. The phoneme to be produced.
  - B. The maximum length of the phoneme string.
  - C. The occurrence of the strobe signal.
  - D. The synthesizer's output volume or level.
14. How many different sounds can be generated using the VOTRAX SC-01?
  - A. 26.
  - B. 40.
  - C. 64.
  - D. 256.
15. In which of the following cases would phoneme inflection most likely be used?
  - A. To change the duration of phonemes.
  - B. To ask a question.
  - C. To generate a stepped sequence of phonemes.
  - D. To produce a pleasant monotone voice.

16. Which of the following is NOT a major step in the unlimited vocabulary phoneme concatenation process?
  - A. Selecting a pre-stored word.
  - B. Determining the entire contents of the phoneme string.
  - C. Determining how to modify the phoneme string.
  - D. Actually modifying the phoneme string.
17. Which mode should be used when the phoneme speech synthesizer needs only a limited vocabulary?
  - A. Output only.
  - B. Limited pitch.
  - C. Pre-stored word.
  - D. Direct phoneme.
18. Which of the following is NOT a recognizable change caused by a punctuation mark.
  - A. An extended pause.
  - B. An emphasized sound.
  - C. A rising inflection.
  - D. A slurring of phonemes.
19. When using a phoneme speech synthesizer such as the SC-01, which of the following statements is false?
  - A. You can give emphasis to a phoneme by repeating it.
  - B. You should avoid silent periods within words, and only use the PA0 and PA1 phonemes between words.
  - C. You should keep your average inflection level in the middle of the synthesizer's range.
  - D. You should program the inflection of important words first, then modify the words as a group.
20. Phoneme data is input to the SC-01's circuitry during which of the following times?
  - A. During the trailing edge of the acknowledge/request signal.
  - B. During the leading edge of the acknowledge/request signal.
  - C. During the leading edge of the strobe pulse.
  - D. During the trailing edge of the strobe pulse.



## EXAMINATION ANSWERS

For your convenience, the page where the correct answer can be found is shown following the answer.

1. C — A phoneme. [9-22]
2. C — The spectrograph. [9-30]
3. D — /s/. [9-17]
4. D — A stop. [9-18]
5. B — The process of shaping speech using the tongue, teeth, and lips. [9-16]
6. D — Is impossible. [9-14]
7. A — An unlimited selection of words or sentences. [9-36]
8. B — The duration of the phoneme produced. [9-38]
9. C — At a data rate of approximately 70 bits per second. [9-38]
10. A — Voiced. [9-40]
11. A — An external timing signal at 720 kHz. [9-46]
12. C — Change the 2 most significant bits of the 8-bit phoneme data code. [9-61]
13. D — The synthesizer's output volume or level. [9-48]
14. D — 256. [9-47]

- 15. B — To ask a question. [9-60]
- 16. A — Selecting a pre-stored word. [9-65 to 9-72]
- 17. C — Pre-stored word. [9-36]
- 18. D — A slurring of phonemes. [9-65 to 9-72]
- 19. B — You should avoid silent periods within words,  
and only use the PA0 and PA1 phonemes between  
words. [9-70]
- 20. C — During the leading edge of the strobe pulse. [9-47]

*Unit 10*

**ET-18 INTERFACING**

## CONTENTS

Introduction .....	10-3
Unit Objectives .....	10-4
Unit Activity Guide .....	10-5
Interfacing Fundamentals .....	10-6
Interfacing With Random Access Memory .....	10-24
Interfacing With Read Only Memory .....	10-41
Interfacing With Control Circuits .....	10-55
Experiments 16, 17, and 18 .....	10-64
Unit Examination .....	10-65
Unit Examination Answers .....	10-70

## INTRODUCTION

As you learned in previous units, a robot consists of many systems and subsystems that work in conjunction with one another to perform a given task. For example, a sensing system would be rather useless unless the information provided by the sensors were used to perform a specific task, even monitoring. By the same token, the robot requires permanent and temporary storage locations for the large amount of data it must handle.

In robot applications, the interaction between the various input and output systems, random access memory, and read only memory, is the most important function performed by the MPU controller. For this reason, we have devoted an entire unit to the interfacing requirements necessary to control a robot. Using the ET-18 Robot Trainer as your model, you will study how these major systems are "tied together" to perform specific tasks.



## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. State the primary differences between RAM and ROM.
2. Determine which types of devices can receive data from, and input data to, the ET-18 MPU controller.
3. Identify various 3-state logic buffers.
4. Describe the function of address lines, data lines, decoders, latches, and buffers.
5. Explain the function of the 6808 microprocessor R/W, VMA, and E control lines.
6. Interface between the MPU and two RAMs using schematic diagrams of the ET-18.
7. Interface between the MPU and ROM using schematic diagrams of the ET-18.
8. Interface between input and output control circuits using schematic diagrams of the ET-18.
9. Describe how ROMs, PROMs, EPROMs, and EEROMs differ.
10. Explain the operation of a RAM storage cell.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "Interfacing Fundamentals."	_____
<input type="checkbox"/> Answer Programmed Review questions 1-12.	_____
<input type="checkbox"/> Read "Interfacing With Random Access Memory."	_____
<input type="checkbox"/> Answer Programmed Review questions 13-18.	_____
<input type="checkbox"/> Read "Interfacing With Read Only Memory."	_____
<input type="checkbox"/> Answer Programmed Review questions 19-29.	_____
<input type="checkbox"/> Read "Interfacing With Control Circuits."	_____
<input type="checkbox"/> Answer Programmed Review questions 30-35.	_____
<input type="checkbox"/> Perform Experiments 16, 17, and 18.	_____
<input type="checkbox"/> Complete The Unit Examination.	_____
<input type="checkbox"/> Check The Examination Answers.	_____

## INTERFACING FUNDAMENTALS

Before going into specific ET-18 interfacing examples, we must first discuss some fundamental concepts we will use. First, we will discuss the concept of a bus and the need for 3-state logic. Then we will examine the various control and bus lines of the 6808 MPU, which is the actual controller used with the ET-18. Finally, we will consider the various timing relationships involved in the execution of instructions.

### Buses

In computer jargon, a bus is generally defined as a group of conductors, connected in parallel, over which information is transferred from one place to another. The information can originate from any one of several sources and can be transferred to any one of several destinations. Moreover, on some buses, information can be transferred in either of two directions. These are called bidirectional buses. Of course, for a given bus, only one transfer of information can occur at a time.

Figure 10-1 shows the data bus arrangement in a typical microcomputer application. Generally, in this type of system, all data transfers involve the MPU. Thus data can be transferred in either direction between RAM and the MPU. However, other data transfers are one way only. Data can be transferred from ROM or the input buffer to the MPU. Also, data can be transferred from the MPU to the output latches.

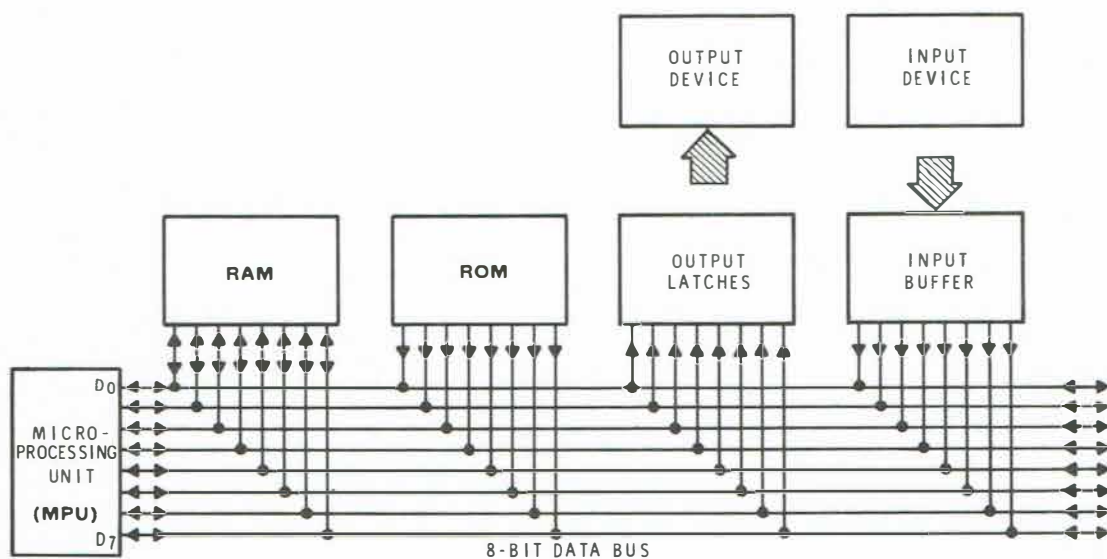


Figure 10-1  
Typical data bus arrangement.

Lets take a brief look at some problems created by this arrangement. First, we must insure that only one data transfer is attempted at any given time. This is done by assigning each destination or source a different address. For example, the RAM, ROM, output latches, and input buffers all have one or more chip enable pins. The proper logic levels on these pins will select or activate the circuit. By assigning each circuit a different address, we insure that only one circuit at a time is enabled.

Figure 10-2 shows the addressing capability added to the block diagram. An address decoder is added for each circuit. The inputs to the address decoders come from the MPU via the address bus. The outputs go to the chip enable lines of the various circuits. Since only one address can appear on the address bus at a given instant, only one of the external circuits will be enabled at a time.

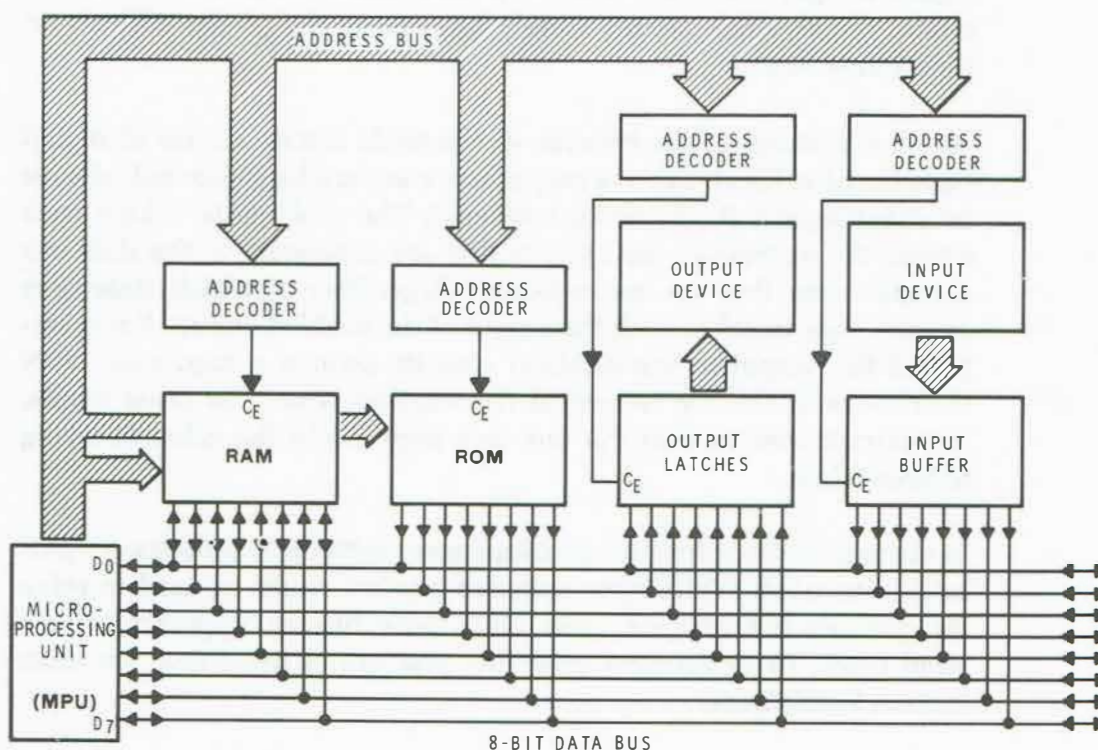


Figure 10-2  
Adding the address capability.

The memories are assigned many addresses since each byte must have its own address. For example, if a  $512_{10}$ -byte RAM is used, it would probably be assigned addresses  $0000_{16}$  through  $01FF_{16}$ . When any one of these addresses appear on the address bus, the RAM is selected via its chip enable line. Notice that a portion of the address bus connects directly to the RAM. This selects the individual byte within the RAM. In the case of the ET-18, two  $2K(2048_{10}$ -byte) RAMs are used; and they are assigned addresses  $0000_{16}$  through  $07FF_{16}$  and  $0800_{16}$  through  $0FFF_{16}$  respectively. Thus, the ET-18 uses a total of  $4096_{10}$  different addresses to communicate with the two RAMs.

In the same manner, the ROM is assigned a range of addresses. If an  $8K(8192_{10}$  byte) ROM is used, as is the case in the ET-18, it may be assigned addresses  $E000_{16}$  through  $FFFF_{16}$ . The ROM must be enabled whenever any of these addresses appear on the address bus. The output latches and input buffers, which will be discussed later, are also assigned unique addresses. Thus, the MPU can communicate with any one of the external circuits simply by placing the proper address on the address bus.

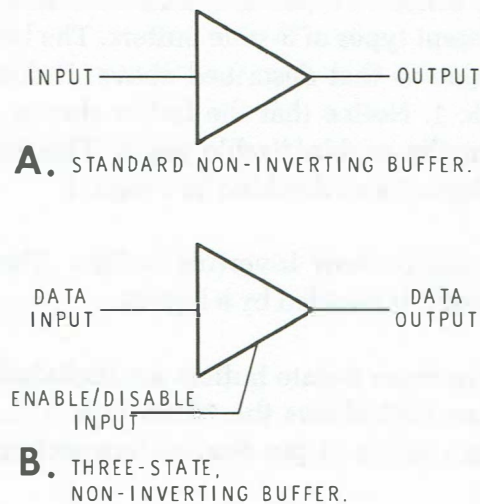
Another problem arises because of the basic 2-state nature of digital logic circuits. Recall that the output of a standard logic gate will always be either logic 1 (high) or logic 0 (low). The problem is: which state should the outputs of the circuits that are connected to the data bus assume when they are not selected? Regardless of which state they assume, they interfere with the output of the enabled circuit. For example, if the output of the disabled circuits assume a high state, they interfere with the low output of the enabled circuit. In other words, one circuit tries to pull the bus line high while the other is trying to force it low.

In the past, to overcome this problem, gates with open collector outputs have been used. While open collector devices could be used to solve this problem in microprocessors, an entirely different approach is most often used. To understand how this problem is overcome, we must discuss **3-state logic**.



## 3-State Logic

As the name implies, 3-state logic devices have a unique third state in addition to the normal 1 (high) and 0 (low) output. Figure 10-3 compares a standard noninverting buffer with a 3-state noninverting buffer.



**Figure 10-3**  
Comparison of standard and 3-state buffers.

Recall that a noninverting buffer increases the current drive of the input signal without changing the logic levels in any way. Thus, the output may be able to drive ten times as many gates as the input. The standard buffer has one input and one output. The output always assumes the same logic level as the input. Because the input must be either 1 or 0, the output must be the same.

By contrast, the 3-state buffer has two inputs. In addition to the normal data input, the buffer has an enable/disable input. This input may be either logic 1 or logic 0 depending upon whether we wish to enable or disable the buffer. The buffer shown in Figure 10-3B is enabled by applying logic 1 to the enable/disable input.

When enabled, the 3-state buffer acts exactly like the standard buffer. The output will assume the same logic level as the data input.

The 3-state buffer is disabled by applying logic 0 to the enable/disable input. When disabled, the output assumes a very high impedance state that is neither logic 1 nor logic 0. While in this high impedance state, the output can be assumed to be disconnected from the rest of the circuit. That is, when the buffer is disabled, its output will not interfere with the circuits to which it is connected.

There are many different types of 3-state devices available. Figure 10-4 shows four different types of 3-state buffers. The buffer shown in Figure 10-4A is the same as that described above. It does not invert, and is enabled by logic 1. Notice that the buffer shown in Figure 10-4B has a small circle at the enable/disable input. This means that the buffer is enabled by a logic 0 and disabled by a logic 1.

Figures 10-4C and D show inverting buffers. The first is enabled by a logic 1; the second is enabled by a logic 0.

Generally, four or more 3-state buffers are included in a single integrated circuit. Figure 10-5 shows the 74126 type TTL IC. It contains four 3-state buffers in a single 14-pin dual-in-line package.

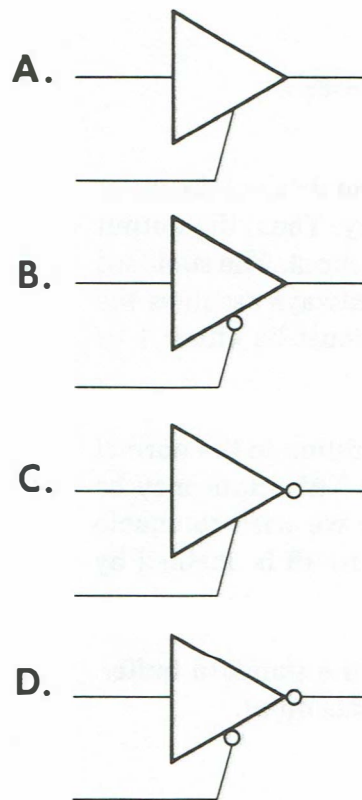


Figure 10-4  
Four types of 3-state buffers.

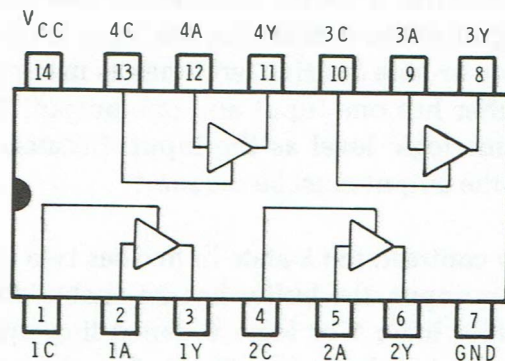
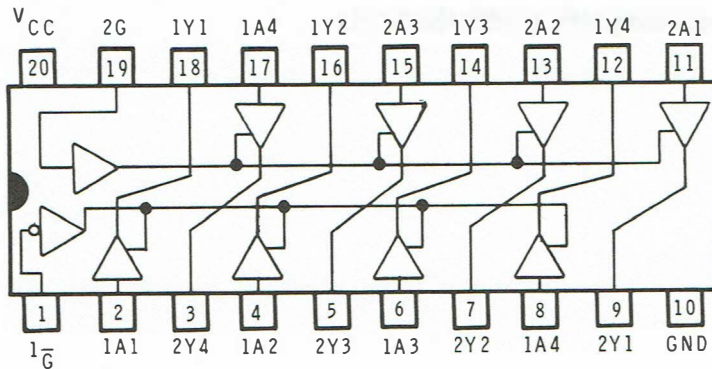
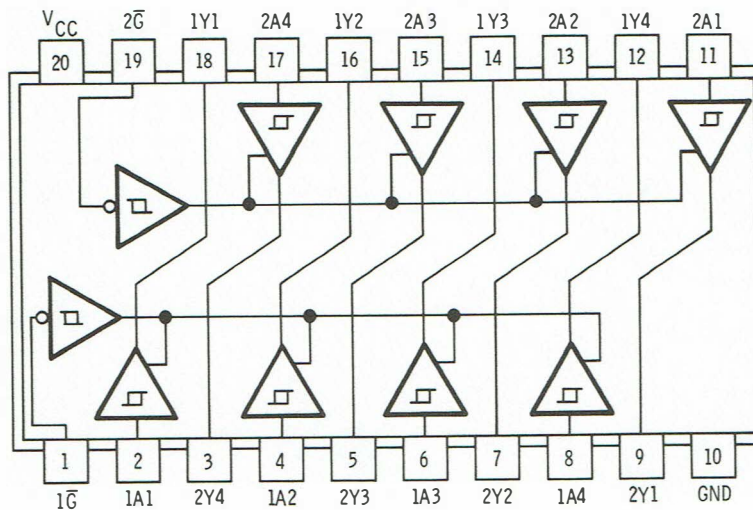


Figure 10-5  
The 74126 IC contains four 3-state buffers in a single 14-pin package.

Figure 10-6A shows eight 3-state buffers in a single 20-pin package. The four lower buffers are enabled by a logic 0 at pin 1 of the IC. The four upper buffers are enabled by a logic 1 at pin 19. Another version of the same type device used in the ET-18 is shown in Figure 10-6B. In this case, the four lower and four upper buffers are enabled when a logic 0 is applied to pins 1 and 19 respectively. Buffers of this type are often called octal buffers, bus extenders, line drivers, etc., depending on how they are used.



(A)



(B)

Figure 10-6

Typical 3-state buffers.

(A) The 74LS241 contains eight 3-state non-inverting buffers.

(B) The 74LS244 used in the ET-18.

While many different forms of 3-state buffers are available, many microprocessor support circuits do not require separate 3-state buffers. Most RAMs and ROMs have their own 3-state buffers built in. Thus, any time the RAM or ROM is not selected, it automatically goes to its third state. In this state, the outputs are said to be off, disconnected, disabled, floating, or in their high impedance state.

The ET-18, however, has several input/output devices such as motion detection, speech, light and sound sensing, etc., that do not have their own 3-state buffers built in. Thus, buffers are required to make these I/O devices compatible with the MPU.

## The MPU Interface Lines

Before you can interface a microprocessor to its support circuits or to the outside world, you must become familiar with its pin assignments, control lines, etc. Figure 10-7 shows the pin assignments and typical interface connections of the 6808 MPU, which is used as the controller for the ET-18. Notice that the MPU is in a single 40-pin dual-in-line package.

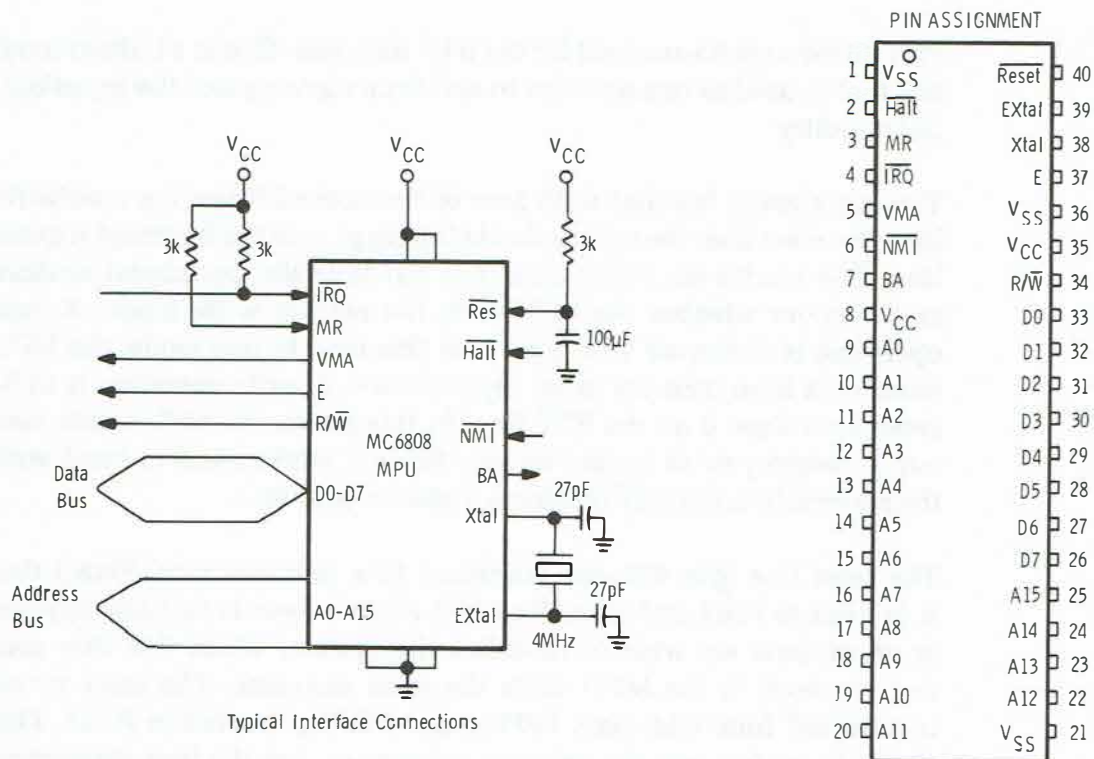


Figure 10-7  
6808 pin assignment and interface connections.



The 6808 microprocessor has a total of five pins for power. Pins 8 and 35, labeled Vcc, are connected to a +5-volt supply. Pins 1, 21, and 36 are labeled Vss and connected to ground.

Pins 9 through 20 and 22 through 25 make up the 16-bit address bus. These pins connect to 16 3-state output drivers in the MPU. Each driver is capable of driving one standard TTL load. When disabled, the address lines act as open circuits. This capability is sometimes used to allow another device to gain control of the address bus. In this way, some external device can address memory. This is referred to as "direct memory access" (DMA).

Pins 26 through 33 are used for the 8-bit data bus. This is a bidirectional bus that is used to transfer data to and from memory and the input/output circuitry.

You are already familiar with four of the control lines: the read/write line, the reset line, the nonmaskable interrupt, and the interrupt request line. The read/write ( $R/\overline{W}$ ) line (pin 34) tells the peripheral devices and memory whether the MPU is in the read or write mode. A read operation is indicated by a logic 1 on this line. In this mode, the MPU reads data from memory or an input device. A write operation is indicated by a logic 0 on the  $R/\overline{W}$  line. In this mode, the MPU sends data out to memory or an output device. Since it works hand-in-hand with the address bus, the  $R/\overline{W}$  line has a 3-state capability.

The reset line (pin 40) was discussed in a previous unit. Recall that it is used to  $\overline{\text{reset}}$  and start the MPU when power is initially applied or at anytime we wish to initialize the system. When this line goes to logic level 1, the MPU starts the reset sequence. The reset vector is retrieved from addresses  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$ , located in ROM. This vector is loaded into the program counter so that the first instruction is fetched from that address. This capability is used to direct the MPU to the start of the monitor or control program. In the ET-18, the reset sequence is initiated only when power is initially applied to the robot or through manual keyboard entry.

The nonmaskable interrupt ( $\overline{\text{NMI}}$ ) line (pin 6) was mentioned when interrupts were discussed. A high-to-low transition on the  $\overline{\text{NMI}}$  line will initiate the nonmaskable interrupt sequence. The high-to-low transition forces the MPU to pick up the  $\overline{\text{NMI}}$  vector at addresses  $\text{FFFC}_{16}$  and  $\text{FFFD}_{16}$ . This vector, again in ROM, is the address of the  $\overline{\text{NMI}}$  service routine. Depressing the “abort button”, located on the experimental board, is the only action that will initiate  $\overline{\text{NMI}}$ , in the ET-18. The  $\overline{\text{NMI}}$  service routine will then halt all ET-18 functions, immediately. It should be noted that no other ET-18 function can override  $\overline{\text{NMI}}$ .

The interrupt request ( $\overline{\text{IRQ}}$ ) line (pin 4) was also discussed earlier. While this is similar to the nonmaskable interrupt, there are three fundamental differences. First, the  $\overline{\text{IRQ}}$  signal is maskable; it will be ignored if the interrupt mask bit is set. Second, the  $\overline{\text{IRQ}}$  sequence picks up the  $\overline{\text{IRQ}}$  vector from addresses  $\text{FFF8}_{16}$  and  $\text{FFF9}_{16}$ . Third, the  $\overline{\text{IRQ}}$  line is level sensitive. That is, a logic 0 on pin 4 causes the interrupt sequence. Compare this to the  $\overline{\text{NMI}}$  line which requires a logic 1-to-0 transition.

Any one of 8 different functions, in the ET-18, can cause an interrupt to the CPU. These functions are:

1. An output from the sonar receiver telling the CPU that an object has been detected.
2. An output from the motion detector informing the CPU that the robot has detected some type of motion.
3. An output from the power supply indicating to the CPU that the batteries that power the **motors** are low, and need to be recharged.
4. An output from the power supply indicating to the CPU that the batteries that supply power to the **logic circuitry** are low, and need to be recharged.
5. An input from the “real time” clock which is distributed throughout the system, and is used as a reference for various functions.
6. A trigger input from the remote “hand-held” teaching pendant which informs the CPU that the teaching pendant has control of the robot. When the teaching pendant trigger is released, all remote programming instructions to the robot cease.

7. An input from the optical reflector module which tells the CPU how much distance the main drive wheel has traveled.
8. An input from the experimental board which informs the CPU that a student wired experiment has generated an interrupt.

The valid memory address (VMA) line (pin 5) is an output. It indicates to peripheral devices that the address on the address bus is a valid one. This signal is necessary because occasionally a false address will appear on the address bus. The VMA signal is used to disable peripheral devices when the address is not valid. A logic 1 at pin 5 indicates that the address is valid and that the peripheral devices may respond accordingly. A logic 0 indicates that the address is not valid and that the peripheral devices should ignore the address. The VMA line is used in any decoding scheme.

The  $\overline{\text{HALT}}$  line (pin 2) provides a hardware method of halting MPU operation. If the input is forced low, the MPU will finish its present instruction, then it will halt. When halted, all 3-state lines go to their off state. This effectively disconnects the MPU from the address and data buses. This line is sometimes used to implement single instruction operation. By controlling the  $\overline{\text{HALT}}$  line, the MPU can be forced to stop after each instruction is executed. This can be a valuable aid in troubleshooting hardware and debugging programs. In many applications, the ET-18 being one, the halt capability is simply not required. In this case, the  $\overline{\text{HALT}}$  line is permanently connected to +5 volts, logic 1.

Another control line, the bus available (BA) line (pin 7) indicates whether or not the MPU is executing instructions. As you know, the MPU may stop executing instructions for either of two reasons. First, the WAI instruction will cause the MPU to stop until an interrupt is received. Or, the MPU can be stopped by forcing the  $\overline{\text{HALT}}$  line low. A logic 0 on the bus available line indicates that the MPU is running. Conversely, a logic 1 indicates that the MPU has stopped. When the MPU is stopped, all 3-state outputs go to their off state. Thus, the MPU is effectively disconnected from the buses. The BA line is not used in the ET-18.

The 6808 has an internal oscillator that may be crystal controlled. The crystal is connected in a parallel resonant configuration between pins 38 and 39. These are designated as Xtal and EXtal on Figure 10-7. A divide-by-four circuit has been fabricated within the 6808. Thus, a 4 MHz crystal connected to pins 38 and 39 results in a 1 MHz output. This output (pin 37) is designated as "E." The 1 MHz Enable (E) output supplies the clock for peripheral devices.

The final control line of the 6808 is labeled memory ready (MR). This input (pin 3) is a control signal that allows stretching of the 1 MHz E output. When MR is high, E will be normal pulse width. When MR is low, E may be stretched multiples of half periods. This feature allows for interfacing to slow memories. In the ET-18, MR is tied high, +5 volts, therefore, E is always a normal 1 MHz clock signal.

## Instruction Timing

Before going further, the timing relationship between the various control and bus signals will be discussed. The discussion will start with the most basic timing relationship: the timing for a single instruction.

Figure 10-8 shows the 2-phase clock signals required by the microprocessor. Keep in mind that the 6808  $\phi 1$  and  $\phi 2$  clock signals are generated internally but controlled by an external crystal. It should be noted that the  $\phi 2$  clock signal is the enable (E) output of the 6808. These two clock signals control every single action that takes place in the MPU and its peripheral devices. To illustrate this, we will discuss the significant events that occur during the fetch and execution of the LDAA immediate instruction.

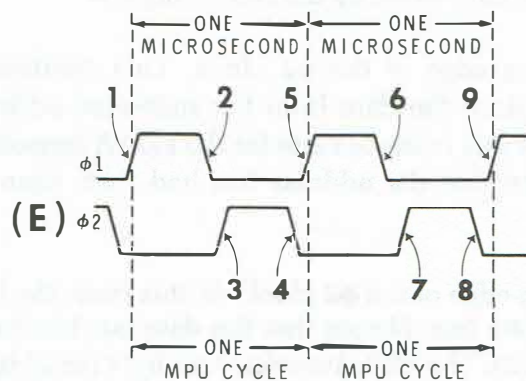


Figure 10-8  
Timing for the LDAA immediate instruction.



Recall that this is a 2-byte instruction. The first byte is the opcode ( $86_{16}$ ). The second byte is the number that is to be loaded into accumulator A. This instruction requires two MPU cycles. During the first cycle, the opcode is fetched and decoded. During the second cycle, the operand is retrieved from memory and is placed in accumulator A.

Notice that the significant events occur at the edges of the clock pulses. Assume that, prior to time 1, the program counter contains the address of the LDAA immediate instruction.

At time 1, the address is transferred from the program counter to the address bus via the memory address register. Notice that this occurs at the positive-going edge of the  $\phi 1$  clock. If the VMA and  $R/\overline{W}$  lines are not already at logic 1, they will be switched to logic 1. A logic 1 on VMA indicates to memory that this is a valid address. A logic 1 on the  $R/\overline{W}$  indicates to memory that the MPU wishes to read the byte at the indicated address.

Time 2 is the falling edge of the  $\phi 1$  clock. At this time, the program counter will be incremented by one, to the address of the next byte in memory. However, this will not change the address on the address bus. Remember that this address is latched into the memory address register. Thus, the output address is still that of the first byte of the LDAA instruction.

The events which occur during the  $\phi 1$  clock are initiated within the MPU itself. In fact, in most systems, the  $\phi 1$  clock is applied only to the MPU. The  $\phi 2$  clock, or (E), on the other hand, is applied to the peripheral circuits as well as the MPU. Thus, the RAMs, ROMs, and other I/O devices are controlled by the enable (E) line.

Time 3 is the rising edge of the  $\phi 2$  clock. This positive-going edge forces memory to place the data from the indicated address onto the data bus. Recall that this is the opcode for the LDAA immediate instruction, or  $86_{16}$ . Notice that the address has had from time 1 to time 3 to stabilize.

Time 4 is the falling edge of the  $\phi 2$  clock. At this time, the MPU accepts the byte from the data bus. Notice that the data bus has had from time 3 to time 4 to stabilize. The MPU transfers this byte ( $86_{16}$ ) to the instruction register where it is decoded and interpreted as an LDAA immediate opcode. This tells the MPU that the next byte in memory is the operand that is to be loaded into accumulator A.



This completes the first MPU cycle. During this cycle, the opcode was simply read from memory and decoded. This corresponds to the fetch phase discussed earlier for our hypothetical MPU. Notice that an MPU cycle corresponds to one cycle of the clock. Now let's see what happens during the second cycle, or the execute phase of the instruction.

At time 5, the address of the operand is transferred from the program counter to the address bus. At time 6, the program counter is incremented by one in anticipation of the next fetch phase.

At time 7, the rising edge of the  $\phi 2$  clock causes the operand to be transferred from memory to the data bus. At time 8, the operand is latched into the MPU where it is transferred to accumulator A. This completes the second MPU cycle and the execution phase of the instruction. Time 9 represents the start of the fetch phase for the next instruction. The LDAA immediate instruction required two MPU cycles or two cycles of the clock. Assuming a 1 MHz clock, two microseconds are required for this instruction. However, in actuality, the ET-18 uses a 3.579545 MHz crystal to control the internal MPU oscillator. Therefore, because of the internal divide-by-four circuit, the clock pulses are at a frequency of 894.886 KHz which, in turn, equates to 1.1 microseconds for each cycle.

## Timing of Program Segment

Now that you are familiar with the timing of a single instruction, several instructions will be put together to form a sample program. You can then study the timing relationship between the bus signals, clock signals, the  $R/\overline{W}$  line, etc.

The sample program is shown in Figure 10-9. Using the immediate addressing mode, it loads  $07_{16}$  into accumulator A and adds  $21_{16}$ . The result is then stored in location  $0001_{16}$ . Notice that the first instruction resides at address  $0010_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ HEX CONTENTS	COMMENTS
0010	86	LDAA#	Load Accumulator A immediate with
0011	07	07	07.
0012	8B	ADDA#	Add to Accumulator A immediate
0013	21	21	21.
0014	97	STAA	Store the result
0015	01	01	at this address.
0016	...	...	Next instruction

Figure 10-9  
Sample program segment.

Figure 10-10 illustrates the timing relationships. At the top the  $\phi 1$  and  $\phi 2$  (E) clock signals are shown. The information that appears on the buses and control lines for each clock period is shown at the bottom. This program segment requires eight clock or MPU cycles which are numbered one through eight. Next, you will see what happens during each of these cycles.

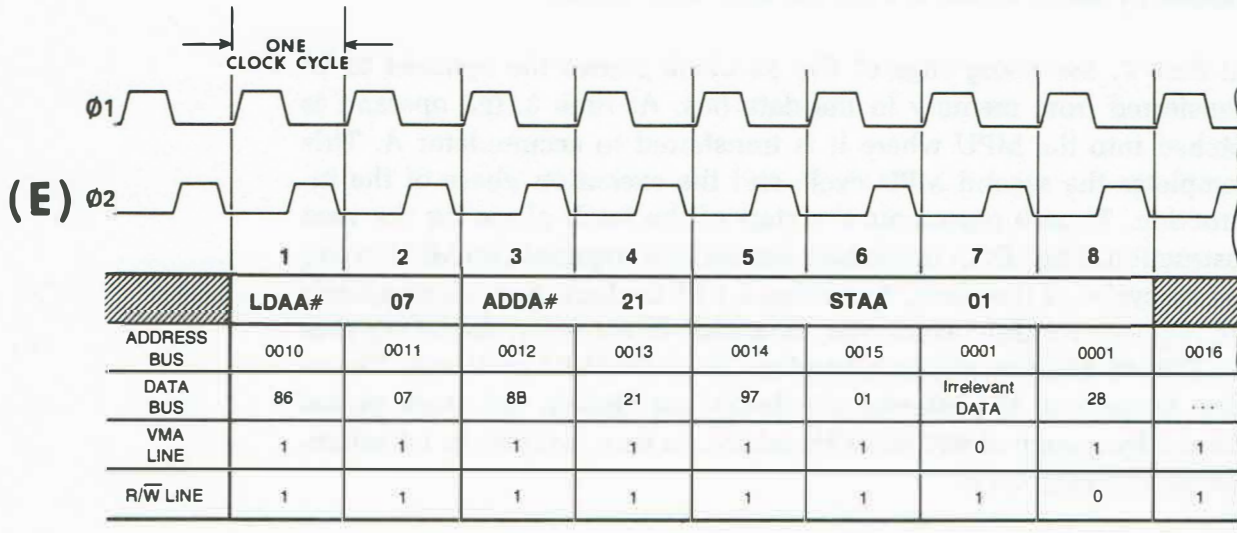


Figure 10-10  
Timing of a sample program segment.

**Cycle 1.** During the first cycle, the address of the LDAA# instruction (0010<sub>16</sub>) is placed on the address bus. As a result, the opcode 86<sub>16</sub> is read from the address and is picked up by the MPU. Since this was a read operation from a valid address, the VMA and R/W lines are at logic 1. The MPU decodes the opcode and recognizes that this is an LDAA# instruction. Consequently, it knows that the next byte in memory is the operand that is to be loaded into the accumulator. During this cycle, the program counter was incremented to 0011<sub>16</sub> so that it now points at the operand.

**Cycle 2.** This is the execution phase of the LDAA# instruction. The address of the operand (0011<sub>16</sub>) is placed on the address bus. The operand (07<sub>16</sub>) is read out on the data bus and is placed in accumulator A. In the process, the program counter is incremented to 0012<sub>16</sub>. This completes the execute phase of the first instruction.

**Cycle 3.** This is the fetch phase of the next instruction. The address  $0012_{16}$  is placed on the address bus. The opcode at that address is read out and placed on the data bus. The MPU picks up the opcode, decodes it, and discovers that it is an  $\text{ADDA\#}$  instruction. In the process, the program counter is incremented to  $0013_{16}$ .

**Cycle 4.** Here, the address  $0013_{16}$  is transferred to the address bus and the selected memory location is read out. Therefore, the operand  $21_{16}$  is placed on the data bus. The operand is picked up by the MPU and is added to the contents of the accumulator. The sum  $28_{16}$  is retained in accumulator A. The program counter is incremented to  $0014_{16}$ .

**Cycle 5.** This is the fetch phase for the third instruction. The address  $0014_{16}$  is placed on the address bus. The opcode for  $\text{STAA}$  is read out and decoded. The MPU recognizes that the direct address mode is used. Thus, it will interpret the next byte in memory as the address at which the sum is to be stored. The program counter is incremented to  $0015_{16}$ .

**Cycle 6.** Address  $0015_{16}$  is placed on the address bus. Notice that the contents of this location is the address at which the sum is to be stored. Thus,  $01$  is read out on the data bus where it is picked up by the MPU. Because the MPU recognized that direct addressing is indicated, it assumes that the address at which the sum to be stored is  $0001_{16}$ . This address is retained in the MPU. The program counter is incremented to  $0016_{16}$ .

**Cycle 7.** During this cycle, the MPU prepares to store the sum. To do this, it must transfer the address  $0001_{16}$  to the address register. Also, it must transfer the sum from the accumulator to the data register. While this is happening, the MPU must refrain from all external data transfers. To prevent unwanted data transfers, the MPU switches the VMA address line low. This tells all peripheral devices that the address is not a valid one and that no data transfers should be initiated. Thus, the peripheral devices simply ignore the data and address buses for this cycle.

**Cycle 8.** The MPU is now ready to store the sum in memory. The address at which the sum is to be stored ( $0001_{16}$ ) is placed on the address bus. The data to be stored ( $28_{16}$ ) is placed on the data bus. The VMA line is switched high, indicating that this is a valid address. The  $\text{R}/\overline{\text{W}}$  line is switched low, indicating that this is a store, or write, operation. Hence, the sum  $28_{16}$  is stored away in memory location  $0001_{16}$ .

Of course, the computer does not stop here. During the next cycle, the next instruction is fetched and decoded. However, the eight machine cycles illustrated above should give you the idea.

## The 6808 Data Sheets

The preceding section has given you some information on the control lines and timing relationships of the 6808 microprocessor, but you may have some questions that have not been answered here. For this reason, detailed data sheets of this microprocessor are included in Appendix B of this course. It explains in more technical language the capabilities of the 6808 microprocessor. At this point in your studies, you should have little difficulty understanding these data sheets. You may want to refer to these data sheets if you have a question that is not answered in the text.

## Programmed Review

- |   |
|---|
| 1. A bus is a group of conductors connected in _____<br>(series/parallel)<br>over which information is transferred from one place to another. |
| 2. (parallel) A _____ bus can transfer information<br>in either of two directions.  |
| 3. (bidirectional) The transfer of data between the MPU and<br>_____ is an example of a bidirectional data transfer.<br>(RAM/ROM)             |
| 4. (RAM) The MPU selects with individual input and output de-<br>vices through the _____ bus.   |
| 5. (address) 3-state buffers have three possible outputs, logic 1<br>logic 0, and _____ impedance.<br>(high/low)                              |
| 6. (high) Most RAMs and ROMs assume the high impedance state<br>when they _____ been selected.<br>(have/have not)                             |
| 7. (have not) In the ET-18, the R/W line would be at logic<br>_____ if the MPU were reading data from a memory device.<br>(1/0)               |



8. (1) In the ET-18, reset \_\_\_\_\_ be initiated manually.  
(can/cannot)

9. (can) In the ET-18, the NMI routine \_\_\_\_\_ be  
(can/cannot)  
overridden by any other function.

10. (cannot) The VMA signal is used to \_\_\_\_\_  
(enable/disable)  
peripheral devices when there is an invalid address on the address bus.

11. (disable) In some microprocessors, memory ready (MR) is used  
to interface the MPU to \_\_\_\_\_ memories.  
(fast/slow)

12. (slow) In the 6808 microprocessor, the enable (E) output is  
sometimes referred to as the \_\_\_\_\_ clock signal.  
( $\phi 1/\phi 2$ )

( $\phi 2$ )



## INTERFACING WITH RANDOM ACCESS MEMORY

Recall that a RAM is a random access read/write memory. The ET-18 uses the popular static RAM. A static RAM employs bistable flip-flops to store information. Because the data is latched in these flip-flops, no refresh circuitry is required. That is, the contents of the flip-flops will be maintained indefinitely, without updating or refreshing the data, as long as power is applied.

There are many different types of static RAMs available. The smaller static RAMs use the bipolar TTL technique while the more dense RAMs, including the ones in the ET-18, use either MOS or CMOS technology.

### The Static RAM Storage Cell

The basic storage cell for an MOS type static RAM is shown in Figure 10-11. Keep in mind that this cell stores a single bit of data. Thus, each of the 2048 word RAMs used in the ET-18 has 16,384 of these storage cells along with address decoding, data control, and control logic circuitry packaged in a single IC.

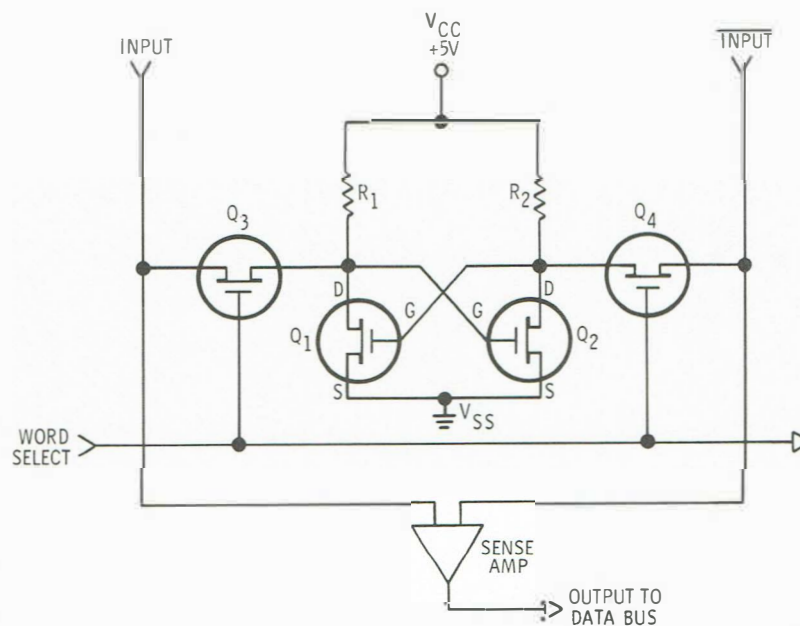


Figure 10-11  
Basic storage cell for a MOS type static RAM.

The storage cell itself consists of four MOS transistors and two resistors. In keeping with current convention, the simplified symbol for the MOS transistor is used. Assume that these are N-channel enhancement type MOSFETs. Recall that an enhancement type MOSFET is normally off, or nonconducting and that, since these are N-channel devices, they can be made to conduct by placing a positive value on the gate terminal. Thus, when the gate is near ground potential, the transistor is off and represents a very high impedance. But, when the gate is high (near  $+V_{cc}$ ), the transistor conducts and has a much lower impedance. If you keep these points in mind, the operation of the cell is easy to understand.

Transistors Q1 and Q2 are cross-coupled so that they form a bistable latch or flip-flop. The bit of data is stored in this latch. R1 and R2 are load resistors for Q1 and Q2 respectively. Q3 and Q4 act as switches which connect input data to the latch during write operations. Also, they connect the data in the latch to the output sense amplifier during read operations. The word select line is connected to the gates of Q3 and Q4. A logic 1 on the word select line will turn Q3 and Q4 on. A logic 0 will keep them turned off.

Since this is a RAM storage cell, we must be able to write into it and to read from it. The following discussion will show how we write into it.

**Write Operation.** Before writing a bit of data into the cell, we must define what constitutes a 1 and 0. Assume that the latch contains a binary 1 when Q2 conducts and Q1 is cut off. Conversely, the latch contains a binary 0 when Q1 conducts and Q2 is cut off.

When power is initially applied, the flip-flop will latch in one condition or the other; however, we can never be sure of which. Data is written into the cell by controlling the input,  $\overline{\text{input}}$ , and word select lines. To store a binary 1, we place a logic 1 (a positive voltage) on the input line. The  $\overline{\text{input}}$  line is always the opposite state of the input line, so it will go to logic 0 (0 volts). The binary 1 is now stored by momentarily applying a logic 1 (positive pulse) to the word select line.

A positive pulse on the word select line will turn switches Q3 and Q4 on. Thus, the positive voltage on the input line is applied through Q3 to the gate of Q2, causing Q2 to conduct. When Q2 conducts, its drain voltage falls to a low value. This reduced voltage is felt on the gate of Q1, cutting Q1 off. When Q1 cuts off, its drain voltage goes high. This increased voltage is felt on the gate of Q2, holding Q2 in the on state.

When the word select line returns to logic 0, Q3 and Q4 cut off. However, the conduction of Q2 keeps Q1 cut off and the high drain voltage of Q1 keeps Q2 conducting. Thus, the binary 1 is latched in the flip-flop and it will remain there until power is removed or until a binary 0 is deliberately written in.

We can later write in a binary 0 by applying a logic 0 to the input, a logic 1 to input, and then pulsing the word select line. When the word select line goes high, Q4 conducts, applying the logic 1 from input to the gate of Q1. This tends to bring Q1 out of cut off. At the same time, Q3 conducts, applying the logic 0 from the input to the gate of Q2. This tends to cut off Q2. As you can see, the flip-flop latches in the opposite state which represents a binary 0.

The sense amplifier and output line are disabled during this time by the MPU read/write line (not shown).

**Read Operation.** The input and input lines are 3-state lines that are enabled or disabled by the read/write line. When the read/write line is high, the input and input lines are effectively disconnected. During this period, we can read data from the storage cell by pulsing the word select line. Assume that the cell is presently storing a logic 1. This means that Q1 is cut off and Q2 is conducting. Consequently, Q1's drain voltage is high (logic 1) while Q2's drain voltage is low. When the word select line swings high, Q3 conducts, connecting the high voltage at the drain of Q1 to the left input of the sense amplifier. At the same time, Q4 also conducts, connecting the low voltage at the drain of Q2 to the right input of the sense amplifier. The sense amplifier interprets this as a logic 1 condition and sets the output line accordingly.

If the flip-flop contains a logic 0 when the word select line swings high, the right input of the sense amplifier receives the higher voltage while the left input receives the lower. The sense amplifier interprets this as a logic 0 and sets the output line accordingly.

## A 128-Word by 8-Bit RAM

Since a storage cell can store only one bit of information, it is easy to see that a large number of these cells are required to form useful memory sizes. For simplicity, we will first discuss the simpler 128-word by 8-bit RAM shown in Figure 10-12. We will then take a look at the 2048-word RAM used in the ET-18. Keep in mind that each one of the squares represents one of the 4-transistor, 2-resistor cells just discussed.

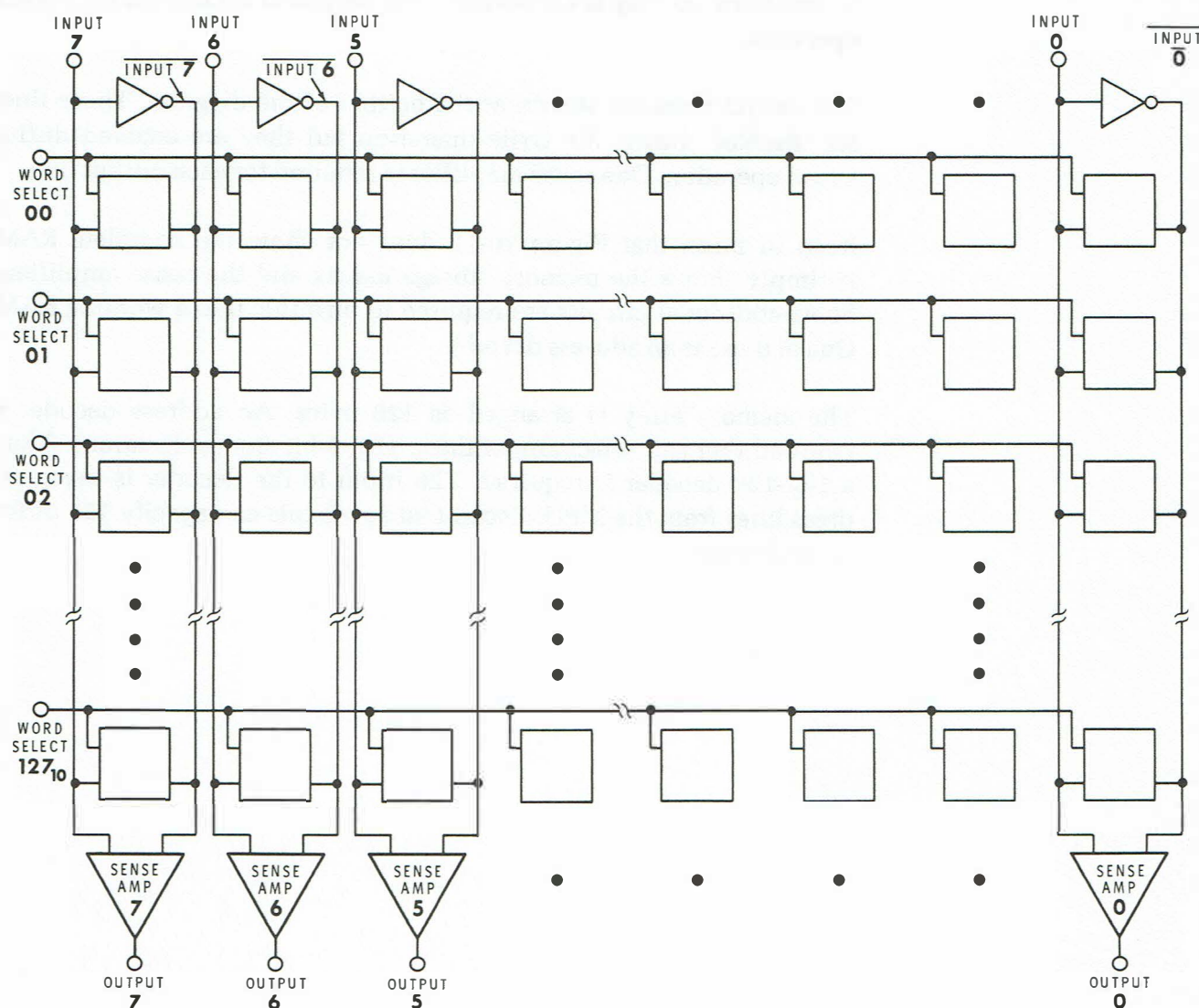


Figure 10-12

Here 1024 storage cells are arranged to form a 128-byte by 8-bit RAM.



The word select lines are shown entering on the left. While only four are shown, there would actually be 128 of these lines — one for each word. Notice that the word select line 00 connects to each of eight storage cells across the top of the figure. In an actual system, these eight storage cells might make up the 8-bit byte we call memory location  $0000_{16}$ .

The input lines are shown at the top of the diagram. For simplicity, only four of the eight lines are shown. Notice that each input line is inverted so that complement inputs can be applied to each cell. Although the details are not shown, both the input and input lines are 3-state lines, so they are effectively disconnected except during a write operation.

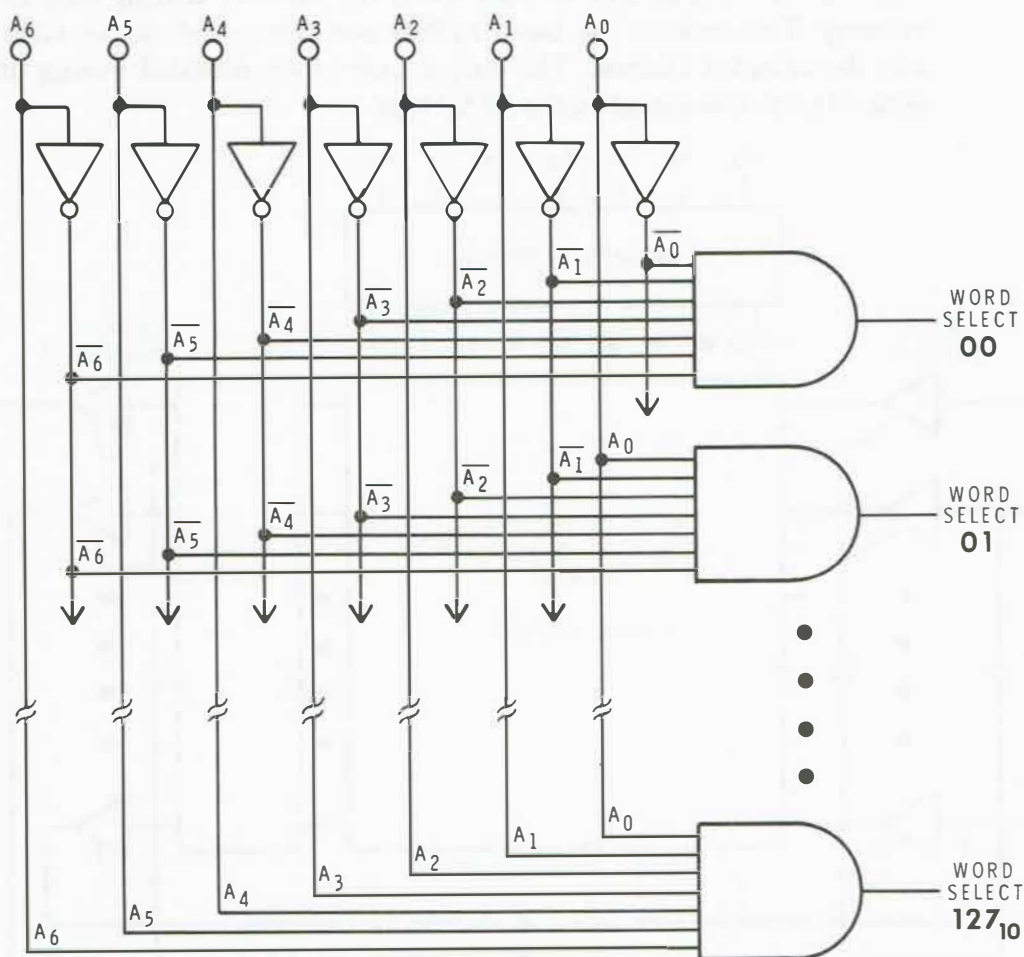
The output lines are shown at the bottom of the diagram. These lines are disabled during the write operation but they are enabled during a read operation. One sense amplifier is required for each output line.

Keep in mind that Figure 10-12 does not show the complete RAM. It simply shows the memory storage matrix and the sense amplifiers. Some additional circuits are required to turn this into a working RAM. One of these is an address decoder.

The memory array is arranged as 128 bytes. An address decoder is required that can select any of these 128, 8-bit storage locations. Thus, a 1-of-128 decoder is required. The input to the decoder is seven address lines from the MPU. Recall that seven bits can specify 128 different addresses.



The address decoder is made up of 128, 7-input AND gates. A simplified diagram is shown in Figure 10-13. Here, only three gates are shown — the first two and the last. Word select line 00 should go high when address lines  $\overline{A_0}$  through  $\overline{A_6}$  are all low. Notice that seven inverters are used to form  $A_0$  through  $A_6$ . Notice also, that these complements are the inputs to the top AND gate. If  $A_0$  through  $A_6$  are all low, then  $\overline{A_0}$  through  $\overline{A_6}$  must all be high. Consequently, the output of the AND gate goes high. Thus, word select line 00 is selected when the low order address is  $0000000_2$ .



**Figure 10-13**  
The 1 of 128 address decoder.

The 01 select line is activated when the address is  $0000001_2$ . Word select line  $127_{10}$  is selected when  $A_0$  through  $A_6$  are all high. Thus it is activated when the address is  $1111111_2$ .

Most RAMs do not have separate input and output lines. Instead they have data lines which can serve either as inputs or outputs. This is possible because the MPU cannot read and write data simultaneously.

Figure 10-14 shows a simplified arrangement of a 128-word RAM using bidirectional data lines. The data lines are shown on the left. The 3-state input buffers are enabled by a high signal on the WRITE line. This line is controlled by the  $R/\overline{W}$  and the chip enable (CE) signal. As you will see, the WRITE line is high when the MPU is writing data into memory. This enables the input buffers and allows data to be written into the selected address. The output buffers are disabled during this period by the low signal on the READ line.

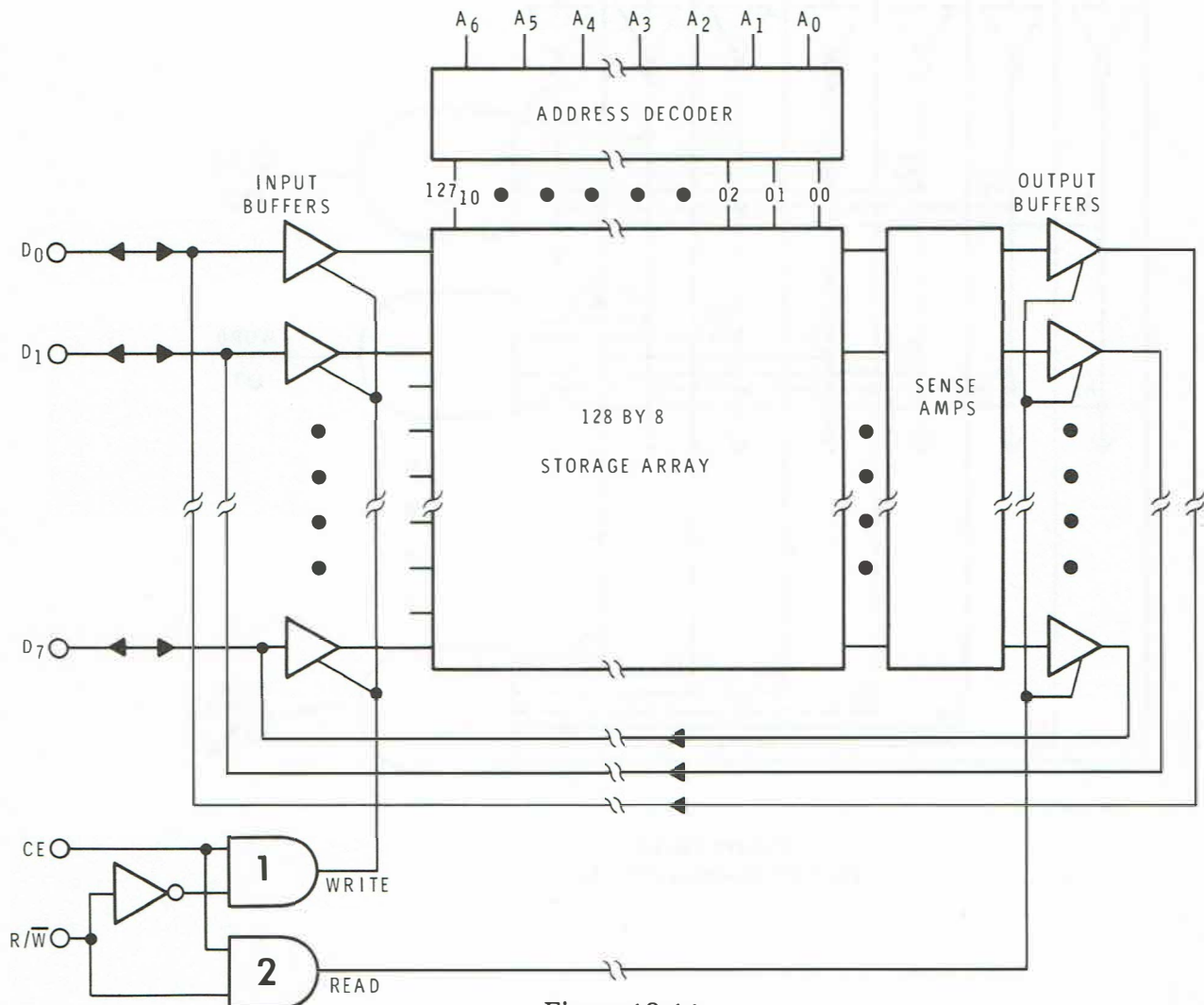


Figure 10-14  
This RAM has bi-directional data lines.

When data is read from the RAM, the READ line is switched high and the WRITE line is switched low. This disables the input buffers and enables the output buffers. Thus, the data at the selected address is read out and placed on the data bus.

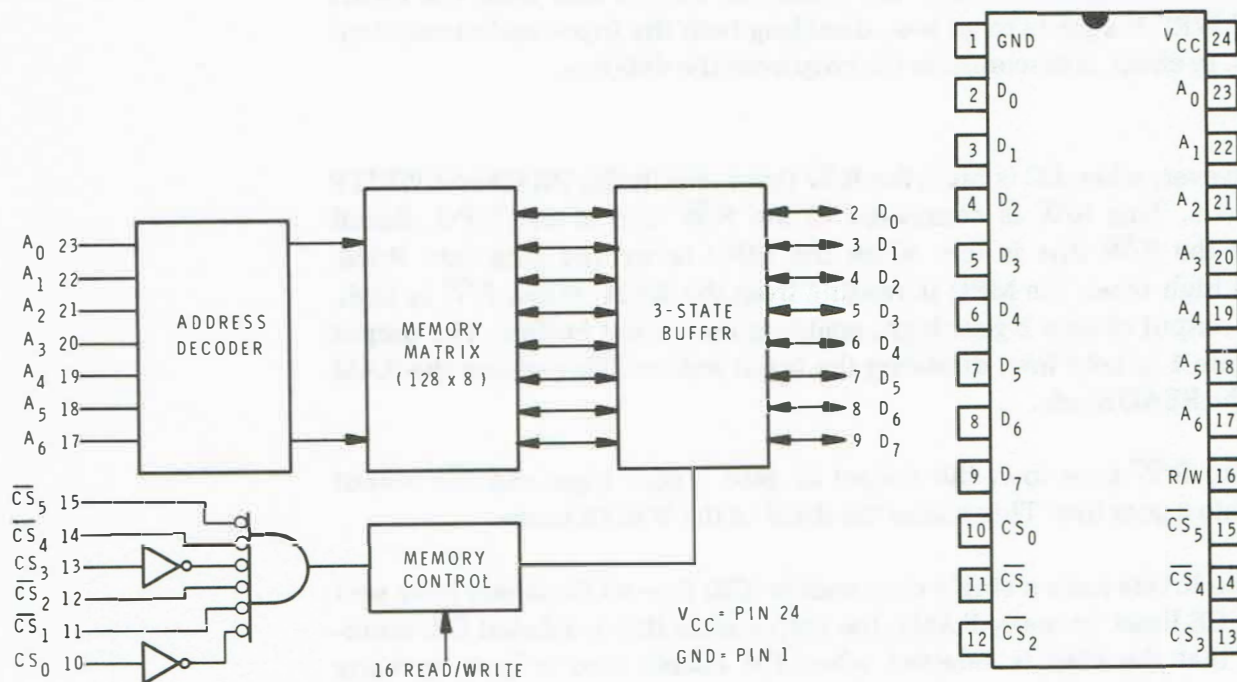
As Figure 10-14 illustrates, the READ and WRITE signals are controlled by the chip enable (CE) signal and the  $R/\overline{W}$  line. The CE input line is switched high when this particular memory chip is selected. If this line is low, gates 1 and 2 are disabled. This causes both the READ and WRITE signals to go low, disabling both the input and output buffers. In effect, it disconnects the chip from the data bus.

However, when CE is high, the  $R/\overline{W}$  line controls the READ and WRITE signals. This  $R/\overline{W}$  is connected to the  $R/\overline{W}$  line of the MPU. Recall that the  $R/\overline{W}$  line is low when the MPU is writing data into RAM, and high when the MPU is reading from the RAM. When  $R/\overline{W}$  is high, the output of gate 2 goes high, enabling the output buffers. The output of gate 1 is held low, disabling the input buffers. This places the RAM in the READ mode.

When  $R/\overline{W}$  goes low, the output of gate 1 goes high and the output of gate 2 goes low. This places the RAM in the WRITE mode.

Some RAMs have a single chip enable (CE) line while others have several CE lines. In many RAMs, the chip enable line is labeled  $\overline{CE}$ , meaning that the chip is selected when the enable line is low. In many instances, they are often referred to as chip select (CS) lines.

Figure 10-15 shows a 128-by-8 RAM that is designed to be used with the 6808 MPU. As shown in the simplified block diagram, this RAM has six chip select lines. The large number of chip selects allows this RAM to be used with little or no external address decoding. As shown in the pin assignment diagram, a 24-pin package is required.



**Figure 10-15**  
The 6810 is a 128-byte by 8-bit RAM.

## 2048-Word by 8-Bit RAM

A simplified diagram of the single IC, high speed, static RAM, used in the ET-18, is shown in Figure 10-16. This is a CMOS 2048-word by 8-bit RAM. The ET-18 has two such RAMs; therefore, it has a total capacity of 4096-words, or 4K, of random access memory.

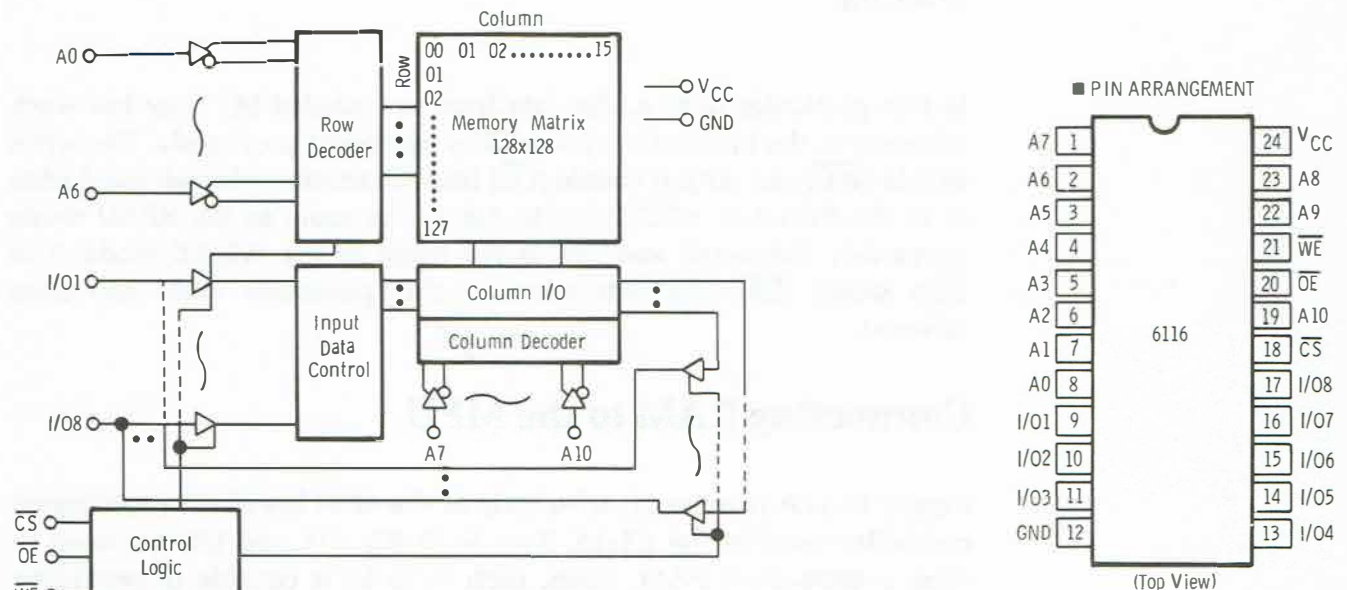


Figure 10-16  
2048-word  $\times$  8-bit high speed static CMOS RAM.

To simplify the address decoders, the cells are arranged in a matrix of 128 rows of 128 cells each, resulting in a total of 16,384 storage cells. ( $128 \times 128 = 16,384$ ). The 128 cells in each row are further divided into 16 columns of 8 bits each. Therefore, we can say the array consists of 128 rows of 16 columns by 8 bits. Thus, each row contains 16 words for a total array capacity of 2048 words. ( $128 \times 16 = 2048$ ).

Two address decoders are used to select the desired word. The row select decoder is a 1-of-128 decoder which uses addresses A0 through A6 to select the desired row ( $00_{10}$  through  $127_{10}$ ). A 1-of-16 decoder is used to select the proper column ( $00_{10}$  through  $15_{10}$ ) within a given row. Four address lines, A7 through A10, are used to specify the column. The selected 8-bit word is at the point where the row and column lines intersect.



For example, assume that the matrix is constructed so that row 00 is at the top of the matrix, row 127 is at the bottom, column 00 is at the left of the matrix, and column 15 at the right. If the contents of address  $11010001010_2$  were desired, it would be found in row 10, column 13. Address lines A0 through A6 which specify the row are contained in the first seven bits of the address ( $0001010_2$ ). This equates to row 10. Address lines A7 through A10 which specify the column are contained in the next four bits ( $1101_2$ ). This equates to column 13. Consequently, if we wanted to examine the contents of row 77, column 11, the address we would send out would be  $10111001101_2$ , or  $05CD_{16}$ .

In this particular device, the data lines are labeled I/O lines but work the same as the bidirectional data lines discussed previously. The write enable ( $\overline{WE}$ ) and output enable ( $\overline{OE}$ ) lines determine whether the device is in the READ or WRITE mode. OE is the same as the READ mode previously discussed and WE is the same as the WRITE mode. The chip select ( $\overline{CS}$ ) line determines if this particular RAM has been selected.

## Connecting RAM to the MPU

Figure 10-17A is a partial schematic of the 6808 based microprocessor controller used in the ET-18. Two 6116 ICs, U4 and U5, are used to form a 4096-word RAM. Thus, each 6116 IC is capable of providing 2K-bytes of memory. Although not shown, the address and data buses from the MPU are connected to a ROM and several other input and output devices. The MPU can communicate with only one of these devices at any one time. To communicate with either of the RAMs the MPU must first select which RAM, by switching its respective  $\overline{CS}$  line low. Before we proceed with the explanation on how we write into or read out of a specific RAM, let's first examine the circuit.

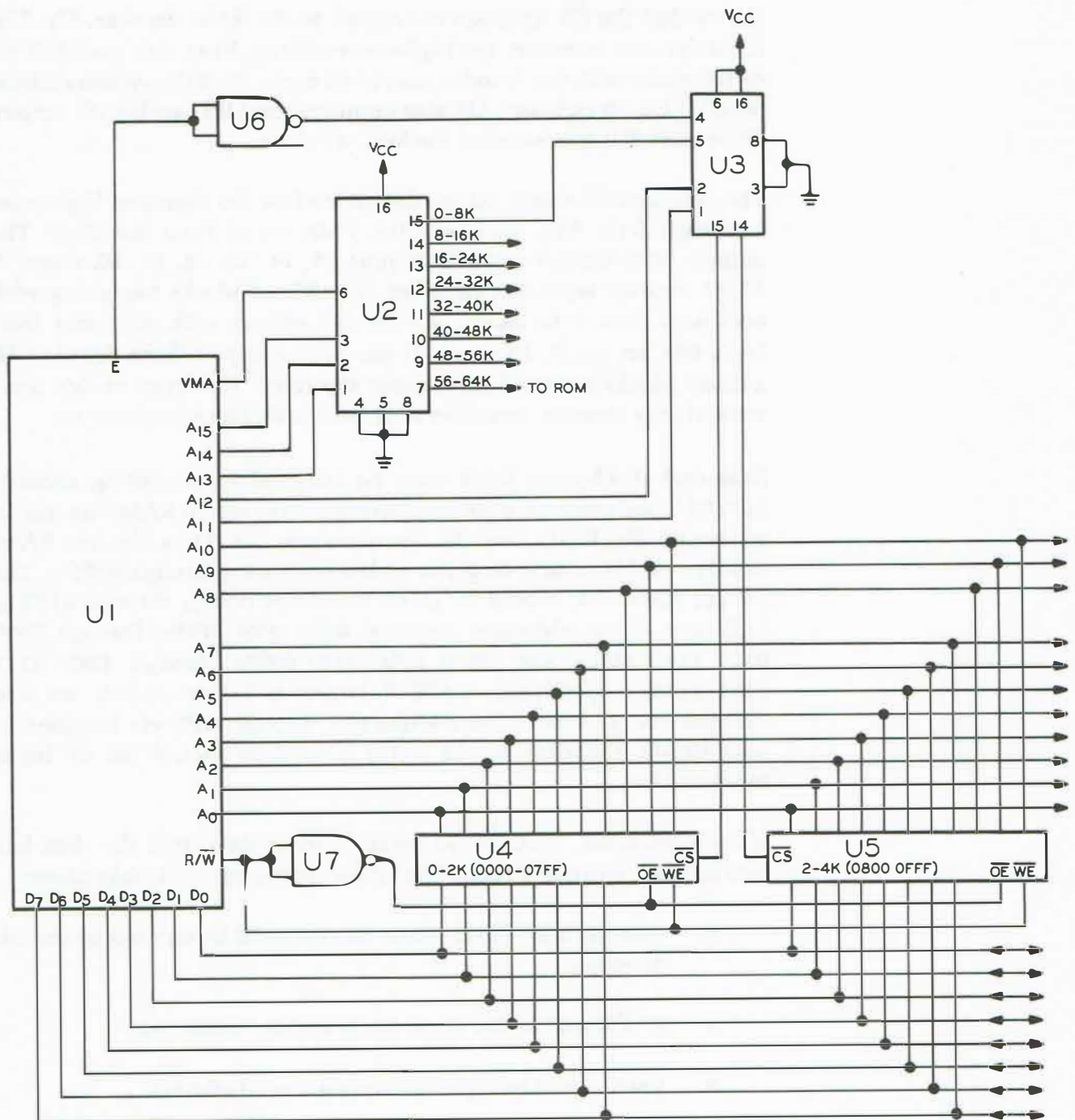


Figure 10-17A  
Partial schematic for the ET-18 MPU, using two 6116  
static RAMs.

Notice that the  $\overline{CS}$  lines are connected to the RAM decoder, U3. The RAM decoder monitors the high-order address lines A11 and A12 directly, along with the decoded output from pin 15 of the system address decoder U2. In addition, U3 also monitors the MPU enable (E) output. Remember, E is the output of the MPU  $\phi 2$  clock.

The outputs of decoder U2 are determined by the inputs of high-order addresses A13, A14, A15, and the VMA signal from the MPU. The outputs from U2 are taken from pins 15, 14, 13, 12, 11, 10, 9 and 7. These outputs represent decoded 8K address blocks beginning with addresses from 0 to 8K on pin 15 and ending with addresses from 56 to 64K on pin 7. As you will see later, many of these decoded 8K address blocks are used throughout the robot. However, at this time, we will only concern ourselves with the 0 to 8K block from pin 15.

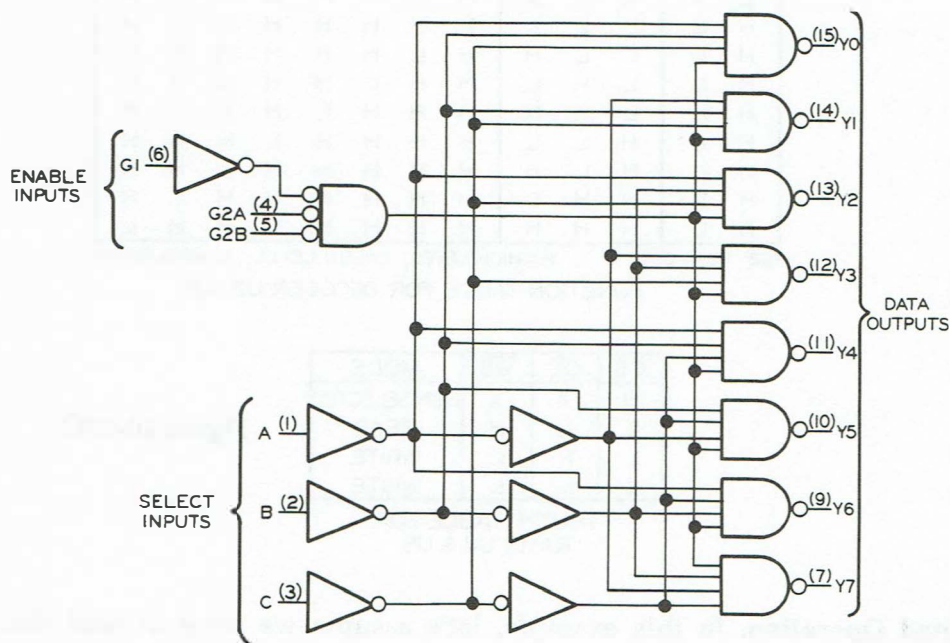
Note each 2048-word RAM must be assigned some starting address. In 6808 based systems, a common practice is to assign RAMs the lowest addresses. The ET-18 uses this common practice. Thus, the first RAM, in this case U4, would be given addresses  $0000_{16}$  through  $07FF_{16}$ . The second RAM, U5, would be given addresses  $0800_{16}$  through  $0FFF_{16}$ . In binary, these addresses are  $0000\ 0000\ 0000\ 0000_2$  through  $0000\ 0111\ 1111\ 1111_2$  and  $0000\ 1000\ 0000\ 0000_2$  through  $0000\ 1111\ 1111\ 1111_2$  respectively. NOTE: Referring to Figure 10-17A, we find that the first 11 bits of the address (A0 through A10) are common to both RAMs. However, bit 12 (A11) is used to control the  $\overline{CS}$  input to U4 and U5.

**Write Operation.** Assume we want to write data from the data bus into memory location  $0327_{16}$ . The following events would take place:

- Address  $0000\ 0011\ 0010\ 0111_2$  would be sent out on the address bus.
- $R/\overline{W}$  would be low since this is a write instruction.
- VMA would be high since this is a valid address.
- E would supply clock pulses to the peripheral devices.

The first 11 bits of the 16-bit address ( $01100100111_2$ ) are felt on address lines A0 through A10 on both RAMs, U4 and U5. The next two bits of the address ( $00_2$ ) are felt on A11 and A12, which in turn, puts a low logic level on pins 1 and 2 of RAM decoder, U3. The last three bits of the 16-bit address ( $000_2$ ) are felt on A13, A14, and A15 which place a low logic level on pins 1, 2, and 3 of system address decoder, U2.

Pins 4 and 5 of decoder U2 are both low, tied to ground, and pin 6 is high since it is tied to VMA. Looking at the logic diagram shown in Figure 10-17B, we find that this condition causes a low output to be felt on pin 15. This low is placed on pin 5 of RAM decoder, U3.



**Figure 10-17B**

Logic Diagram

Pin 3 of U3 is low, tied to ground, and pin 4 of U3 is also low due to the action of NAND gate U6. Pin 6 of U3 is high, tied to Vcc, and pins 1 and 2 are low due to the state of address lines A11 and A12. Looking at the function table for U3, (Figure 10-17C) we find that this condition causes a low-logic-state output on pin 15 and a high output on pin 14. These outputs are applied to the  $\overline{CS}$  inputs of RAMs U4 and U5 respectively.

At the same instant, a low, from the  $\overline{R/\overline{W}}$  output of the MPU is applied to the write enable ( $\overline{WE}$ ) input of both RAMs. Also, through the action of NAND gate U7, a high is applied to the output enable ( $\overline{OE}$ ) pin of both RAMs. Observing the truth table, shown in Figure 10-17C, we see that these conditions cause RAM U4 to be in the write mode and RAM U5 to be not selected. Thus, data from the data bus will be written into memory location  $0327_{16}$  in RAM U4. We will now take a look at a READ operation.

INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	L	H	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

\*G2 = G2A + G2B      H=HIGH LEVEL, L=LOW LEVEL, X=IRRELEVANT

FUNCTION TABLE FOR DECODER U2, U3

CS	OE	WS	MODE
H	X	X	NONSELECTED
L	L	H	READ
L	H	L	WRITE
L	L	L	WRITE

TRUTH TABLE FOR  
RAMS U4 & U5

Figure 10-17C

**Read Operation.** In this example, let's assume we want to read the contents of memory location  $0AD5_{16}$ . The following conditions would be set:

- Address  $0000101011010101_2$  would be sent out on address lines A0 through A15.
- VMA and E would both be high.
- $\overline{R/\overline{W}}$  would be high.



Again, observing Figure 10-17A, we see that the first 11 bits of the address ( $01011010101_2$ ) are felt on address lines A0 through A10 of RAMs U4 and U5. The next two bits ( $01_2$ ) place a high on pin 1 and a low on pin 2 of RAM decoder U3, via addresses lines A11 and A12. The last three bits ( $000_2$ ), on address lines A13, A14, and A15, put pins 1, 2, and 3 of U2 at a low logic level.

Checking the function table shown in Figure 10-17B, we find that the above conditions cause pin 15 of U2 to go low. This low is again applied to pin 5 of U3. Observing the logic levels at the inputs of U3 and looking at the function table, we find the output at pin 14 is low and pin 15 is high. These outputs are again felt on the  $\overline{CS}$  inputs of U4 and U5.

Since we are in the READ mode, the MPU places a high logic on the  $R/\overline{W}$  line. This high is felt on the  $\overline{WE}$  input of both RAMs. Because of NAND gate U7, a low logic level is applied to the  $\overline{OE}$  input of the RAMs. Checking the truth table in Figure 10-17C, we see that RAM U5 has been selected, and it is in the read mode. Thus, data at address  $0AD5_{16}$  will be placed on the data bus.

## Programmed Review

13.	The ET-18 uses a _____ type of RAM. (dynamic/static)
14.	(static) The static type of RAM _____ require refresh circuitry. (does/does not)
15.	(does not) The data in a static RAM _____ be lost when power is removed from the circuit. (will/will not)
16.	(will) A single storage cell can store _____ bit(s) of data.
17.	(one) A 256-word by 8-bit RAM would require _____ storage cells.
18.	(2048) Using Figure 10-16 as our example, an address of (10100010111) would address the contents at row _____ col- umn _____, in the RAM.
(23, 10)	

## INTERFACING WITH READ ONLY MEMORY

In many microcomputer systems, especially dedicated systems such as robotic applications, there is more read only memory, or ROM, than there is read/write memory, or RAM. Recall that the ET-18 has 4K-bytes of RAM available. However, as you will see, the ET-18 has 8K-bytes of ROM. All system related programs and subroutines are stored in ROM. These include routines that start the system, routines used for interrupt servicing, display routines, routines for controlling the robot's actions, etc. In fact, any software that is frequently used should be located in ROM. This is because programs in ROM are easily accessible and are **nonvolatile**, meaning that they are not lost when the system is turned off. As you know, the contents of RAM are lost when power is removed from the system. Obviously, you do not want to reenter a routine in RAM each time the system is powered-up. It is also not desirable to have to load a frequently used routine from a cassette tape each time it is required. With ROM, the frequently used software routines are instantly addressable and available. This is why ROM software is often referred to as **firmware**.

There are a variety of read only memory devices available today. They include mask-programmed ROMs, one-shot programmable ROMs (PROMs), erasable programmable ROMs (EPROMs), and electrically erasable ROMs (EEPROMs). Each of these devices has its particular advantages and disadvantages. In this section, we will briefly discuss these various ROMs and take a close look at the 68A364 mask-programmed ROM used in the ET-18. As each type of device is discussed, keep in mind that the application will clearly determine the type of ROM to be used.

### Mask-Programmed ROM

Mask-programmed ROM is always programmed by the ROM manufacturer. This type of ROM is available with "canned" programs such as system monitors, editors, interpreters, etc. Also, many of the business program for personal computers and popular electronic game packages are available in mask-programmed ROM. When this is the case, the ROM is usually encased in a plug-in cartridge. You simply plug the cartridge into a compatible system; thus, you can begin using the canned program immediately.

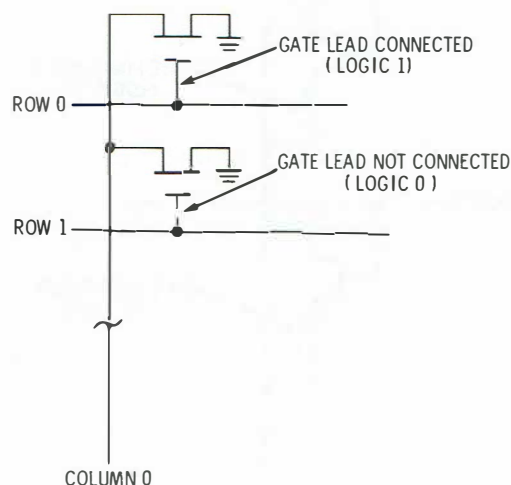
As in the case of the ET-18, you can also specify your own program to be mask-programmed into a ROM. This is called **custom programming**. The ROM device data sheet will specify how your program is to be formatted, and how it should be forwarded to the manufacturer. Most manufacturers require that the program be forwarded to them on punched paper tape, punched cards, or floppy disk. However, you must order many hundreds or even thousands of these custom programmed devices to justify the cost.

In addition to the cost of the ROM device, the manufacturer will charge a one-time mask charge. The mask charge is normally very high; thus, the more custom ROMs you order, the cheaper each individual ROM becomes, since the mask charge is amortized over a larger number of devices. Compared to other ROM devices, the mask-programmed ROM is the most economical, as long as you order several hundred devices.

The biggest disadvantage of mask-programmed ROM is that once it is programmed, the program is fixed and cannot be changed. Any alterations in the program would require a new mask to be made and thus generate a new mask charge.

To program the ROM, the manufacturer makes a **metallization mask** from the user supplied program. The metallization mask is used in the last step of the chip fabrication process to define the 1's and 0's in the memory cells. Two cells of an MOS mask-programmed ROM are shown in Figure 10-18. The ROM cells are determined by rows and columns the same as the RAM cells previously discussed. Each cell of the MOS ROM contains an integrated MOSFET device. The metallization mask will cause the gate lead of the MOSFET to be either connected for a logic 1, or not connected for a logic 0. When the gate lead is connected, the MOSFET can be turned on by applying a potential to the respective row line, which results in a logic 1 level on the corresponding column line. If the gate lead is not connected, the MOSFET cannot be turned on, and no potential, or logic 0, appears on the corresponding column line.

The ROM cells can contain MOSFET devices, as shown in Figure 10-18, or bipolar devices. The advantage of a bipolar ROM is speed. Bipolar ROMs will have access times of between 50 nanoseconds and 150 nanoseconds. On the other hand, MOS ROMs will provide access times between 200 and 400 nanoseconds. However, the bipolar devices are generally more expensive, less dense, and require more power to operate.



**Figure 10-18**  
Two cells of a MOS mask-programmed ROM.



## Programmable ROM or PROM

The PROM is **user programmable**, meaning that you program it yourself rather than have the manufacturer do it. You are not faced with a mask charge, but the cost on a per unit basis is relatively high compared to a large mask-programmed ROM purchase. In addition, you must have a PROM programming unit, or **PROM burner**, to program the device. The PROM burner is actually a small microcomputer that allows you to enter your program in its RAM and then transfer that program to the PROM. The PROM burner also supplies the voltage and current levels required to program the device.

Two cells of a bipolar PROM are shown in Figure 10-19. Most PROMs are bipolar devices and, therefore, each memory cell contains an integrated bipolar transistor. The transistor's emitter is connected to the cell column line through a **fuse link**. When the PROM is manufactured, all the fuse links are connected, and thus all cells contain logic 1's. To program the device, the PROM burner programs 0's into the required cells by "blowing" the fuse links. To open the fuse line, a typical PROM will require a programming voltage of 8 to 12 volts and a programming supply current of 400 to 600 mA.

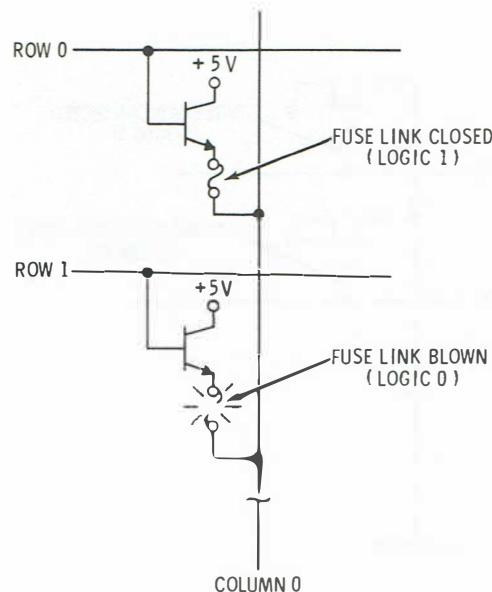


Figure 10-19  
Two cells of a bipolar PROM.

Once the PROM is programmed, it cannot be reprogrammed; if changes must be made to the program, the old PROM must be discarded and a new one programmed. For this reason, PROMs are sometimes referred to as **one-shot ROMs**.

## Erasable Programmable ROM or EPROM

The EPROM is both user-programmable and erasable. It has all the desired features required for early system development and provides non-volatile data storage. Unlike the PROM, an EPROM can be reprogrammed many times. For this reason, EPROMs are becoming much more popular than PROMs.

An EPROM is pictured in Figure 10-20. It is easy to recognize by the quartz window above its integrated circuit, or chip. The quartz window is provided so that the chip can be exposed to ultraviolet light for erasing. Exposing the window to a high intensity ultraviolet light source for approximately 30 minutes is all that is required to erase any information stored in the device.

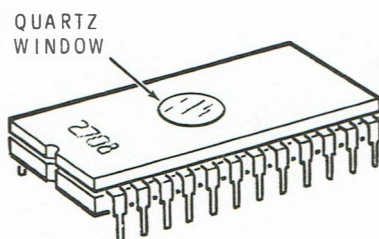


Figure 10-20  
A typical EPROM device, the 2708.

Each EPROM memory cell contains a MOSFET device with a **floating gate**. Figure 10-21 shows a cross-section and schematic symbol of this device. The floating gate is completely surrounded with silicon dioxide, an excellent insulating material. To program a cell, a relatively high potential of approximately 25 volts is applied to the control gate and drain leads. At the same time, the source and P-type substrate are held at a ground, or 0 V, potential. The difference in potential within the device causes electrons to penetrate the silicon dioxide insulating layer around the floating gate. Once the electrons penetrate the insulating layer, they have nowhere to go and become trapped by the floating gate. The floating gate therefore becomes charged, and the cell is programmed. The charge will remain on the floating gate for many years, because there is very little leakage through the surrounding silicon dioxide layer. A charge decay curve is shown in Figure 10-22. The curve shows that even after a 10-year period, over 80% of the floating charge still remains.

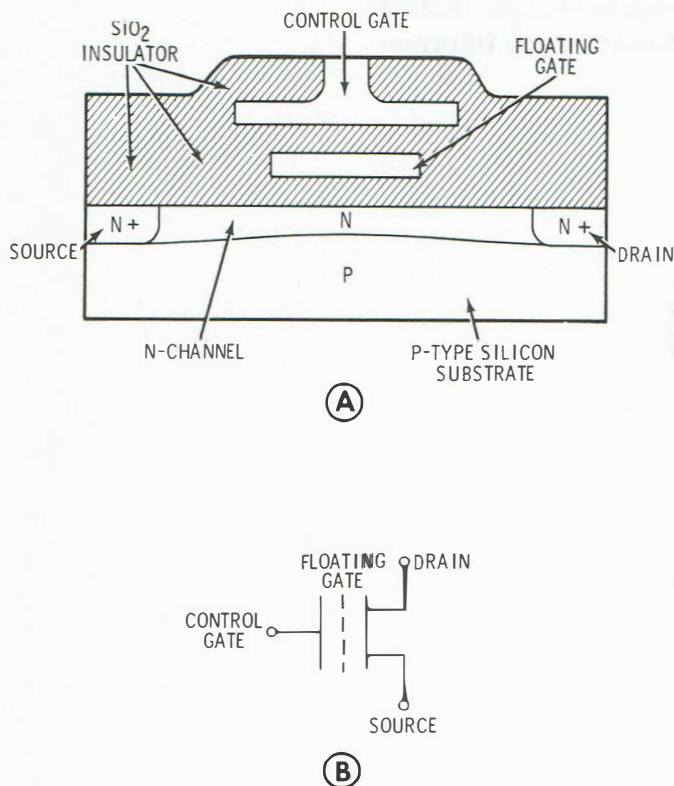


Figure 10-21  
EPROM memory cell cross section (A) and schematic symbol (B).

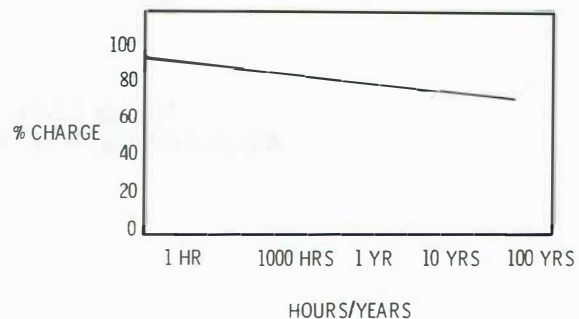


Figure 10-22  
Charge decay of a typical EPROM.

You can eliminate the floating charge gate and erase the cell by exposing the silicon dioxide insulating layer to ultraviolet light. During the exposure period, the insulating layer loses much of its insulating properties and becomes more conductive, allowing the floating gate charge to leak away. In addition, the ultraviolet light source excites the floating gate electrons and gives them more energy to leak through the silicon dioxide material. As stated earlier, it takes approximately 30 minutes to erase an EPROM with a high intensity ultraviolet light source. You can also erase an EPROM by exposing it to sunlight or normal room lighting, but it takes approximately 1000 times longer to erase it properly.

Since EPROMs are relatively easy to program and even easier to erase, they are being used extensively for system development and nonvolatile data storage in small systems. When developing a commercial system, the system software is stored and **debugged** using EPROMs. Once the designer is satisfied that there are no more bugs in the software, mask-programmed ROMs are ordered to replace the EPROMs. Most manufacturers produce pin-for-pin compatible mask-programmed ROMs for direct replacement of the EPROMs in their product line.

## Electrically Erasable ROM, or EEROM

The last type of ROM that we will discuss is the electrically erasable ROM, or EEROM. This type of ROM is also called E<sup>2</sup>ROM and **electrically alterable ROM, or EAROM**. A more generic term for these devices is **read-mostly memory**. They can actually be classified as read/write memories, but the write cycle is very long compared to the read cycle, thus the term read-mostly memory.

With the proper software, hardware gating, and timing signals, EEROMs can be programmed and erased within the host microcomputer system. However, they can not be used as general purpose read/write memory since the write cycle takes at least one millisecond and the erasure period is at least 100 milliseconds. Thus, it would require 101 milliseconds to change the contents of a memory location. But, the read cycle access time is typically 400 nanoseconds. This is sufficient to allow the EEROM to be used as a ROM type device.

The structure of the EEROM memory cell is similar to that of an EPROM cell. Each cell contains a floating gate N-channel MOSFET device that is programmed by storing a charge on the floating gate and erased by applying a strong current pulse to neutralize the stored charge.

The advantage of using EEROMs is that they do not have to be removed from the system to be programmed or erased. However, they have several disadvantages. Most EEROM devices are relatively low density devices. In addition, they are expensive when compared to other ROM devices and are not very reliable after many erasures. They also cannot retain data as long as EPROM devices. For these reasons, the EEROM is not widely used. However, as processing technology improves, EEROM devices will probably be used in more industrial and commercial microcomputer systems.

A comparison summary of the different ROM devices just discussed is provided in Figure 10-23. Next, you will study a specific ROM device, the 68A364, and how it is used in the ET-18.

Device	User-Programmed	Reprogrammable	Density	Cost
Mask-programmed ROM	NO	NO	Very High	Lowest in large quantities
PROM	YES	NO	Low	Medium
EPROM	YES	YES	High	Medium
EEROM	YES	YES	Low	High

Figure 10-23  
ROM device comparison table.



## The MCM68A364 Mask-Programmed ROM

The 68A364 is a 24-pin,  $8192 \times 8$ -bit, mask-programmed ROM, used in the ET-18. It has a typical access time of 350 nanoseconds. This ROM is easy to use, since it operates from a single +5V power supply, is TTL compatible, and needs no clock. A block diagram with pin assignment is provided in Figure 10-24.

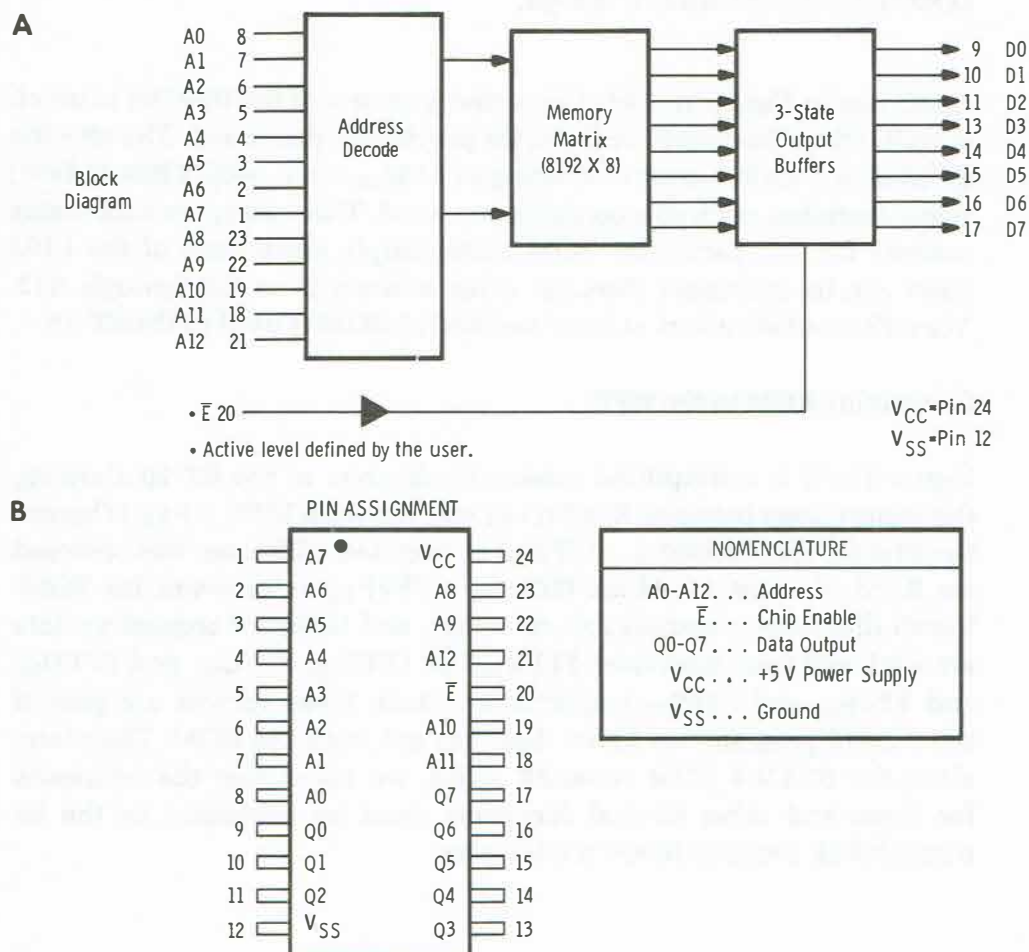


Figure 10-24  
The ET-18 68A364 8192  $\times$  8-bit mask-programmed ROM.

You can always determine the memory capacity of a ROM device by counting its address and data pins. Notice in Figure 10-24A that the 68A364 has 13 address pins, labeled A0 through A12, and 8 data pins, labeled D0 through D7. Thus, the 68A364 is a  $2^{13} \times 8 = 8192 \times 8$ , or  $8K \times 8$ , ROM. In addition to the address, data, and power pins, the 68A364 has a chip enable, or  $\bar{E}$ , pin.

The active level of the chip enable input and the memory content is defined by the user. The chip enable input deselects the output and puts the ROM in the power-down mode. In the case of the ET-18, the ROM is deselected when  $\bar{E}$  is high.

As shown in Figure 10-24A, the memory matrix of the 68A364 is much simpler than the matrix of the RAM previously discussed. The 68A364 ROM uses a matrix array consisting of  $8192_{10}$  rows, each 8 bits (1 byte) wide; therefore, each row contains one word. This makes the addressing scheme for this particular ROM quite simple since each of the 8192 rows can be addressed directly, using address lines A0 through A12. We will now take a look at how the 68A364 ROM is used in the ET-18.

### Connecting ROM to the MPU

Figure 10-25 is a simplified schematic diagram of the ET-18 showing the connections between ROM (U-6) and the 6808 MPU (U-1). Whereas the first 8K block ( $0000_{16} - 1FFF_{16}$ ) of decoded addresses was reserved for RAM, the last 8K block ( $E000_{16} - FFFF_{16}$ ) is reserved for ROM. Recall that reset, nonmaskable interrupt, and interrupt request vectors are retrieved from addresses  $FFFE_{16}$  and  $FFFF_{16}$ ,  $FFFC_{16}$ , and  $FFFD_{16}$ , and  $FFF8_{16}$  and  $FFF9_{16}$  respectively. Since these vectors are part of the control program, we know that they are stored in ROM. Therefore, since the 68A364 ROM is an 8K ROM, we know that the addresses for these and other control functions must be contained in the 56 through 64K decoded block of addresses.

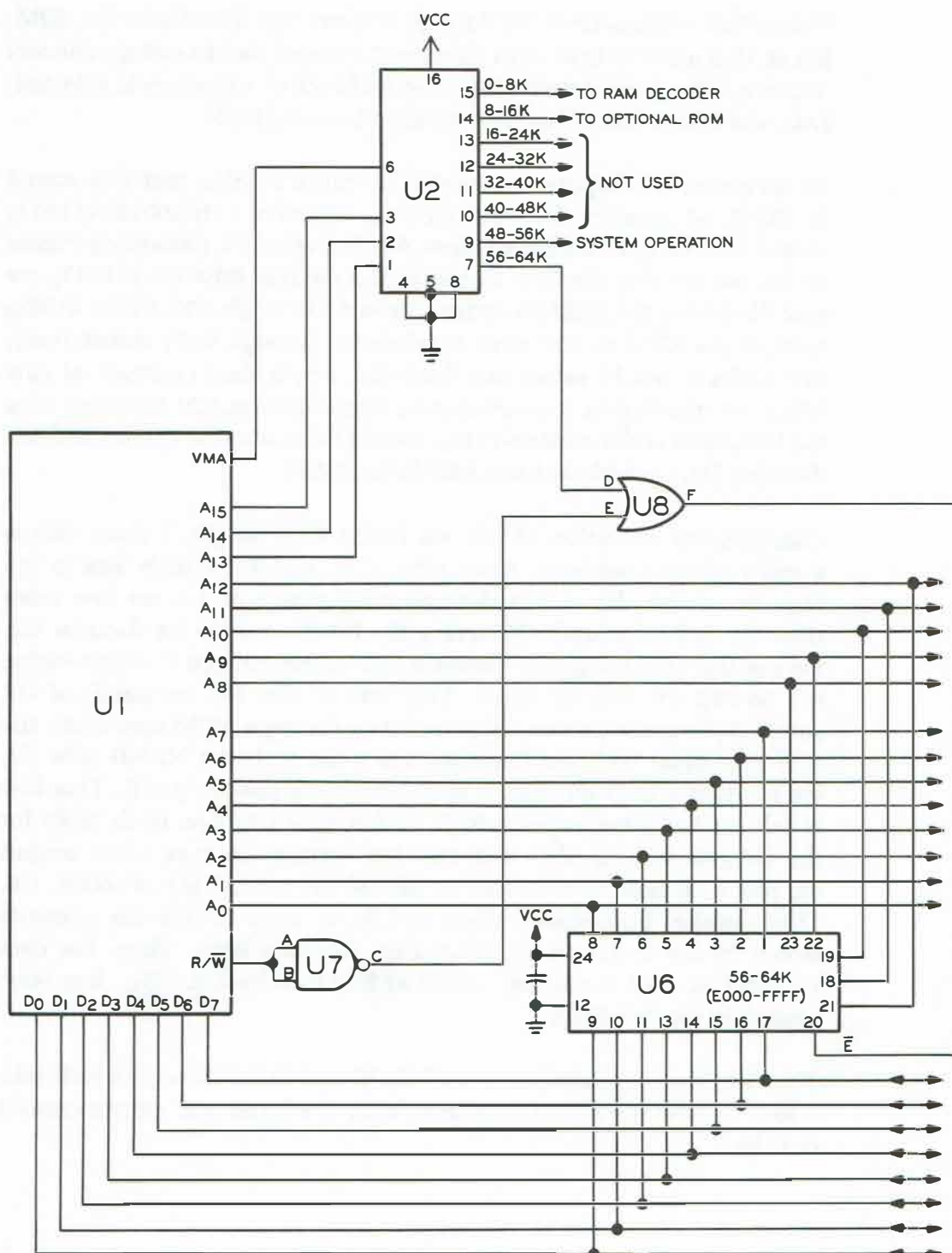


Figure 10-25  
ET-18 ROM to MPU interface.

Notice that address lines A0 through A12 are tied directly to the ROM. Recall that address lines A13 through A15 were tied to system address decoder, U2, which determines what 8K block of addresses is selected. Lets now take a look at how data is taken from the ROM.

As an example, suppose we wanted to start a routine that was stored in ROM, at memory location E17B<sub>16</sub>. Address 1110000101111011<sub>2</sub> would be sent out on address lines A0 through A15. Observing Figure 10-25, we see that the first 13 bits of the address 0000101111011<sub>2</sub> are sent directly to the ROM on address lines A0 through A12. (If the 8192<sub>10</sub> rows in the ROM matrix were numbered 0 through 8191 respectively, this address would select and load the 1-byte data contents of row 379<sub>10</sub> into the ROM's internal 3-state output buffers.) At the same time the last 3 bits of the address (111<sub>2</sub>) would be sent to the system address decoder, U2, via address lines A13 through A15.

Checking the condition of U2, we find a high on pin 6 since this is a valid memory address. Also, pins 1, 2, and 3 are high due to the high on address lines A13 through A15; pins 4 and 5 are low since they are tied to ground. Observing the function table for decoder U2, we see that this condition causes a low output on pin 7, representing the 56-64K 8K address block. This low is also felt on pin D of OR gate, U8. Because we can only read data during a ROM operation, the R/W line must be high. Checking the truth table for NAND gate U7, we see that a high on pins A and B causes a low on pin C. This low is felt on the other input, pin E, of OR gate U8. The truth table for the OR gate tells us, that with two low inputs, we have a low output on pin F, which in this case is placed on pin 20 ( $\overline{E}$ ) of ROM, U6. This enables the output buffers which, in turn, causes the contents stored in the buffers to be placed on the data lines. Thus, the data required to start a routine, stored at ROM address E17B<sub>16</sub>, has been placed on the data lines.

Thus far, we have interfaced with RAM and ROM. Now, we will take a look at how the ET-18 interfaces with the input and output control circuits.

## Programmed Review

19.	Programs stored in ROM are considered to be _____, meaning they are not lost when power is turned off.
20.	(nonvolatile) In most dedicated systems, including the ET-18, there is _____ ROM available than there is RAM. (more/less)
21.	(more) The ET-18 uses a _____ type of ROM.
22.	(mask-programmed) When a user specifies a program to be mask-programmed into ROM, it is referred to as a _____ program.
23.	(custom) Once a mask-programmed ROM has been manufactured, the contents of the ROM _____ be altered. (can/cannot)
24.	(cannot) A PROM _____ is a small microcomputer that allows the user to enter a program in its RAM and then transfer that program to a PROM.
25.	(burner) PROMs are sometimes referred to as _____ ROMs.
26.	(one-shot) The _____ is user programmable and can be erased using ultraviolet light.
27.	(EPROM) Exposing the EPROM to sunlight _____ erase its contents. (will/will not)
28.	(will) The EEROM is actually a read/write device, however, the write cycle takes _____ time to complete than the read cycle. (more/less)
29.	(more) The memory capacity of a ROM can be determined by counting the number of _____ and data pins. (address)



## INTERFACING WITH CONTROL CIRCUITS

So far, we have interfaced the ET-18 6808 MPU controller with RAM and ROM. However, in order for the robot to be completely functional, we must also interface the MPU controller to associated input and output control circuitry. Let's look at an actual example of how we interface between the MPU and a specific output control circuit. We will use an output to the experimental board as our example. Since we will frequently be referring to Figure 10-26 and Figure 10-27 throughout this discussion, you may want to remove these two figures from the binder to better follow the sequence of events.

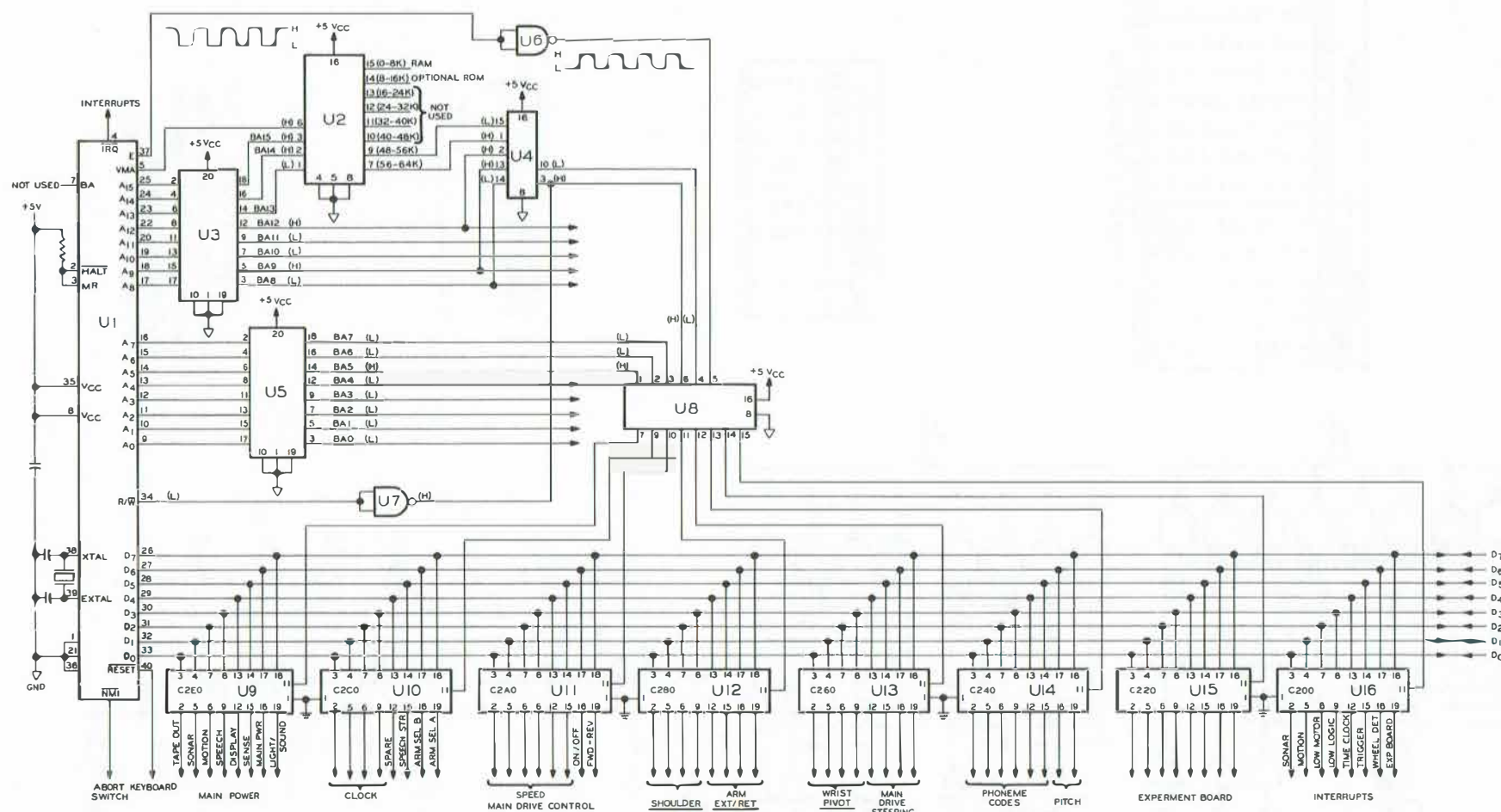
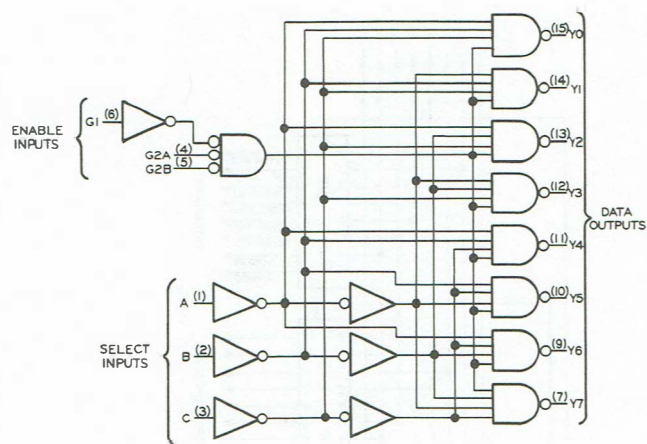
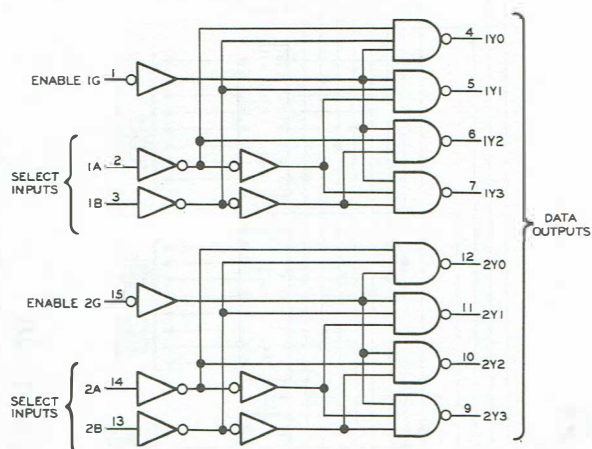


Figure 10-26  
ET-18 CPU and I/O circuitry for outputting data from  
the MPU.



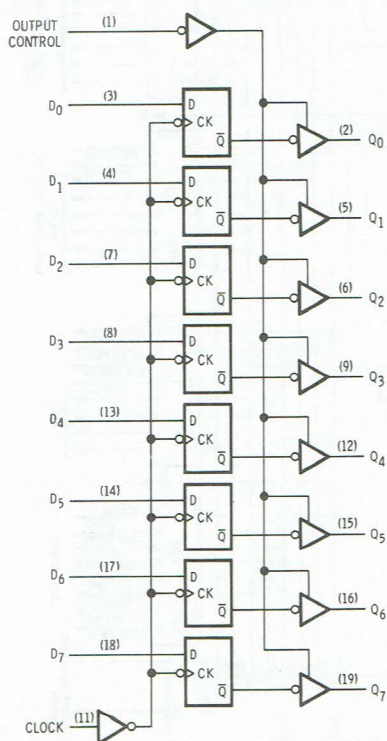
ENABLE		SELECT			OUTPUTS							
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H

\*G2 = G2A + G2B H=HIGH LEVEL, L=LOW LEVEL, X=IRRELEVANT  
FUNCTION TABLE FOR DECODER U2



ENABLE		SELECT			OUTPUTS			
G		B	A		Y0	Y1	Y2	Y3
H		X	X		H	H	H	H
L		L	L		L	H	H	H
L		L	L		L	H	H	H
L		L	L		L	H	H	H
L		L	L		L	H	H	H
L		L	L		L	H	H	H
L		L	L		L	H	H	H

H=HIGH LEVEL, L=LOW LEVEL, X=IRRELEVANT



OUTPUT ENABLE	CLOCK	D	OUTPUT
L	↑	H	H
L	↑	L	L
L	L	X	Q0
H	X	X	Z

Figure 10-27  
Function tables for Figure 10-26.  
(A) Function table for U2 and U8.  
(B) Function table for U4.  
(C) Function table for U9 – U16.

## Outputting Data To The Experimental Board

As you know, it is possible to output data to, and input data from, the experiment board via the MPU. For the first portion of this discussion, we will see how specific data is outputted to the experimental board.

Assume, we wish to output the binary word 10101010 ( $55_{16}$ ) to be used as a control function for an experiment. This can be accomplished by using the simple commands shown below.

```
86
55
B7
C2
20
```

Quite simply, these commands tell the MPU to immediately load accumulator A with the hexadecimal number (55). Next, store the contents of accumulator A, hexadecimal number 55, at address location ( $C220_{16}$ ) which is the output device for the experimental board.

Figure 10-26 shows a portion of the CPU and I/O circuitry used in the ET-18. Notice, when the MPU sends binary command 10101010 ( $55_{16}$ ) to output address  $C220_{16}$ , via data lines D0 – D7, it is also applied to several other output devices simultaneously. Recall, all input and output devices are in parallel with the data lines, therefore, whatever data is felt on one device is felt on all other devices at the same time. Hence, we must have a method ensuring that only the device at address  $C220_{16}$  accepts this data. This is accomplished by the logic state of the microprocessor VMA, E, and  $R/\overline{W}$  lines and the action of decoders U2, U4 and U8.

VMA will be high since  $C220_{16}$  is a valid memory address; also,  $R/\overline{W}$  will be low because the MPU is writing data into an output device. The enable (E) line is alternating low and high logic clock pulses. Address  $C220_{16}$  (1101 0010 0010 0000<sub>2</sub>) is sent out on address lines A0-A15 to decoders U2, U4, and U8 via line buffers U3 and U5.

By now, you should be familiar with decoder operation; however, for your convenience, the logic level of the address lines and decoder inputs and outputs are shown in Figure 10-26. By observing the function tables for U2 and U4, shown in Figures 10-27A and 10-27B, it can be seen that the decoder U8 inputs will be at the logic levels shown in Figure 10-26. (Inputs at pins 1 and 6 are high and inputs at pins 2, 3, and 4 are low.)

The output of U8 is determined by the logic level of the clock (E) pulses applied to pin 5. Let's assume for a moment that the clock pulse (E) applied to pin 5 of U8 is low. Checking the function table for decoder U8, we see that a low applied to pin 5 results in a low output at pin 14 and high outputs at pins 7, 9, 10, 11, 12, 13, and 15. The low at pin 14 is felt at the clock input, pin 11, of U15. U15 is a D-type positive edge-triggered flip-flop used as a temporary storage device for data going to the experimental board. U9 through U14 and U16 are also temporary storage devices for data going to other control circuits in the system. At this instant, U9 through U14 and U16 have a high logic level applied to their respective clock input, due to the outputs of U8.

As shown in Figure 10-27C, a low logic level on the clock input of U15 results in no output from the temporary storage device. The information on the data lines is there; it just cannot be passed through the flip-flops at this time. Temporary storage devices U9 through U14 and U16 also have the same information on their data lines, but, because their clock inputs are high, they simply ignore the data.

However, once the clock enable (E), at pin 5 of U8, starts to go high, it causes the output at pin 14 of U8 to also start to go high. This low to high transition is felt at the clock input, pin 15, of storage device U15. Again checking our function table, shown in Figure 10-27C, we see that this low-to-high transition causes the data on data lines D0 through D7 to be "clocked" through the device to the experimental board. Thus, the information sent out on the data lines to the experimental board, at address location C220<sub>16</sub>, is now available at the experimental board.

Try a few more examples of addressing the different output devices, using the addresses shown on the various devices in Figure 10-26 and the function tables shown in Figure 10-27. Once you feel comfortable using the various addresses, place Figures 10-26 and 10-27 back in the binder.



## Inputting Data From The Sense Circuit

The ET-18 MPU controller makes many decisions based on inputs from several external sources. The input could be a manually entered reset command from the keyboard or a simple switch closure informing the MPU the main drive steering has reached its limit of travel. Also, the input could be an 8-bit binary output from the sensing circuit informing the MPU what level of light or sound it is detecting. For our example of inputting data to the MPU, we will use an 8-bit binary word from the sensing circuit, representing a specific level of light.

Figure 10-28 is a simplified block diagram of the ET-18 sense circuit. The sense circuit accepts inputs from the turret-mounted light-dependent resistor (LDR) and dynamic microphone, digitizes the inputs and makes them available to the MPU.

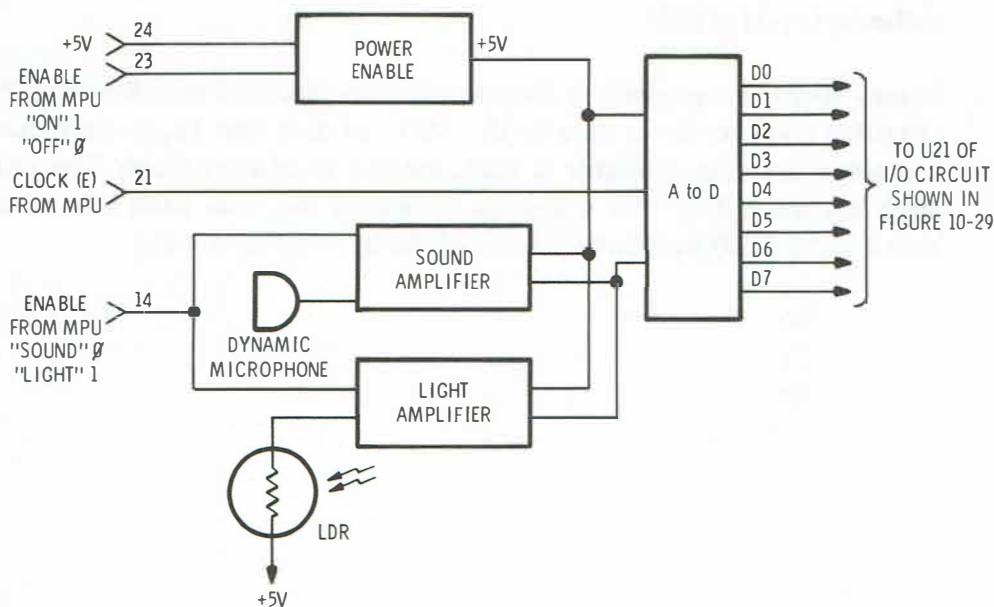


Figure 10-28  
ET-18 sense circuit.

Power to the sense board is controlled by the power enable circuit. When a high level enable signal from the microprocessor is present on pin 23, the power enable circuit turns on. This allows the +5V at pin 24 to be distributed to all circuits on the sense board.

The sound and light sense functions cannot be enabled simultaneously. A signal from the microprocessor at pin 14 determines which of the two functions is selected. A low level at pin 14 selects the sound function; a high level selects the light function. When the level at pin 14 is high, it effectively allows the signal from the LDR to be amplified by the light amplifier circuit. At the same time, the high level signal at pin 14 causes the sound amplifier to shunt the input from the dynamic microphone to ground. The sound signal is thus effectively prevented from passing any further through the system.

The analog output of the light amplifier, representing the amplified light signal, is applied to the analog input of the A-to-D converter. The clock timing for the A-to-D converter is furnished by the (E) clock signal from the MPU. The analog signal from the light amplifier is converted to an 8-bit digital word by the A-to-D converter and made available to the microprocessor. This 8-bit digital word can represent 256 different levels of light.

Figure 10-29 is a simplified diagram showing some of the I/O and CPU circuitry used to input data to the MPU. Notice that input data from the sense board is available at U21, located at address C240. The program shown below will allow us to access the data from the sense board and store it in accumulator A of the MPU for future use.

B6  
C2  
40

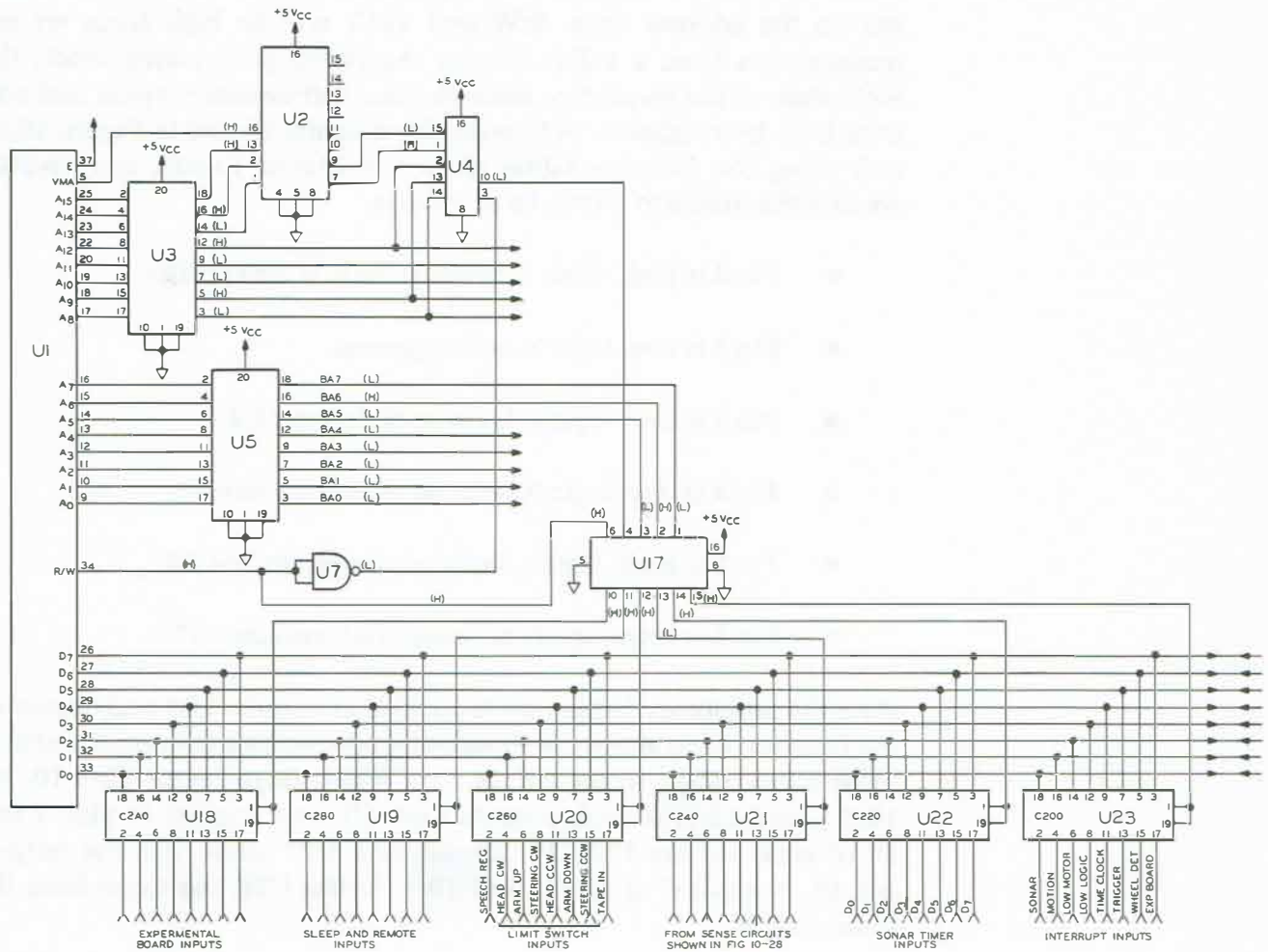


Figure 10-29  
ET-18 CPU and I/O circuitry for inputting data to the MPU.

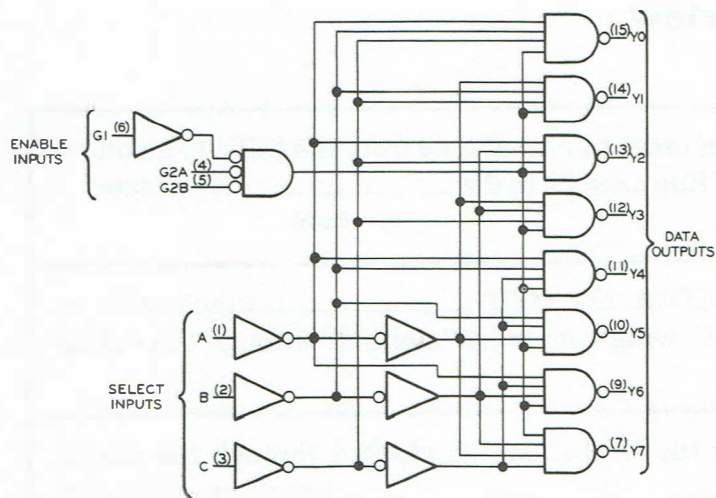
In this example, address  $C240_{16}$  ( $1101\ 0010\ 0100\ 0000_2$ ) will be sent out on the address lines.  $R/\overline{W}$  and VMA will be high since we are reading data from a valid address. Again for your convenience, the logic state of the respective address lines and decoder inputs and outputs have been labeled. Following the diagram shown in Figure 10-29 and using the function tables shown in Figure 10-30A and 10-30B, we find the inputs to U17 to be as follows:

- Pin 6 is high, logic 1, because the  $R/\overline{W}$  line is high.
- Pin 5 is low, logic 0, tied to ground.
- Pin 4 is low, logic 0, because of decoder U4.
- Pin 3 is low, logic 0, because of address line A5.
- Pin 2 is high, logic 1, because of address line A6.
- Pin 1 is low, logic 0, because of address line A7.

With the previously listed inputs applied to decoder U17 and observing the function table shown in Figure 10-30A, we find the outputs of U17 to be a low (logic 0) on pin 14, and highs, (logic 1) on pins 10, 11, 12, 13, and 15. The high outputs from U17 are applied to pins 1 and 19 of octal buffers U18, 19, 20, 22, and U23 while the low output, pin 14, is applied to pins 1 and 19 of buffer U21, the input from the sense circuit.

Noninverting buffers U18 – 23, shown in Figure 10-30C, are configured so that a high on pins 1 and 19 inhibits data from passing through. However, a low on pins 1 and 19 enables the buffer and data is allowed to pass through. Therefore, when a valid **read** address of  $C240_{16}$  is sent out on the address lines, data from the sense circuit will be sent to the MPU.

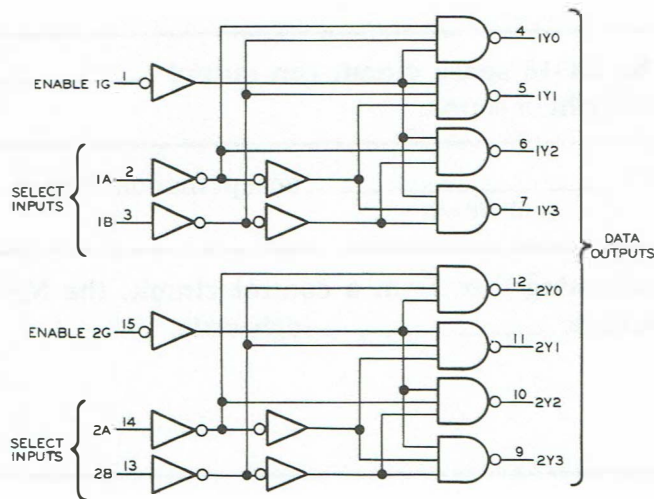
Data from the other input devices can be obtained in a similar manner, by addressing the desired input device.



INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	L	H	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	L	H	H	H
H	L	H	H	L	H	H	H	H	H	L	H	H
H	L	H	H	H	H	H	H	H	H	H	L	H

\*G2 = G2A + G2B H=HIGH LEVEL, L=LOW LEVEL, X=IRRELEVANT

FUNCTION TABLE FOR DECODER U2



INPUTS			OUTPUTS			
ENABLE	SELECT					
G	B	A	Y0	Y1	Y2	Y3
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

H=HIGH LEVEL, L=LOW LEVEL, X=IRRELEVANT

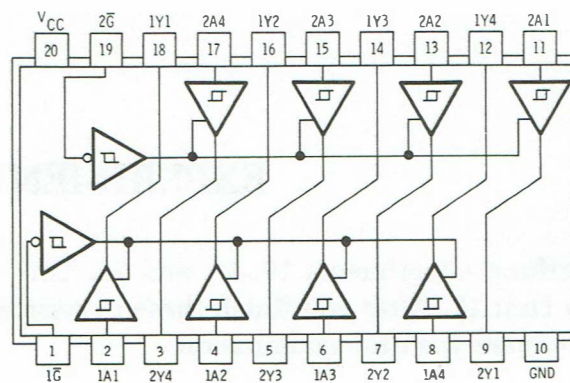


Figure 10-30

Function tables and pin diagram for Figure 10-29.

(A) Function table for U2 and U17.

(B) Function table for U4.

(C) Pin diagram for U18-U23.



## Programmed Review

30.	In the ET-18, in order to output data from the MPU to a control circuit the R/W line must be in the _____ logic state. (low/high)
31.	(low) In the ET-18, the MPU _____ function is responsible for allowing data to be "clocked" through the output device.
32.	(enable, E) In the ET-18, data is clocked through the output device on the _____ to _____ transition of the enable line.
33.	(low, high) The ET-18 sense circuit can output _____ different levels of light or sound.
34.	(256) The MPU _____ accept manual inputs. (will/will not)
35.	(will) When inputting data from a control circuit, the MPU R/W line will be in its _____ logic state.  (low/high)
	(high)

## EXPERIMENTS

Perform experiments 16, 17, and 18. You will find these experiments in Unit 12. After you finish these experiments, return to this unit and complete the Unit examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct answers that appear after the exam.

1. Which of the following ET-18 devices has the capability of transferring data to, and receiving data from, the MPU?
  - A. Random Access Memory.
  - B. Read Only Memory.
  - C. Main drive steering limit switch.
  - D. Abort switch.
  
2. To ensure that only one input or output device is enabled at a given time:
  - A. All devices are assigned the same address.
  - B. Input and output devices are located on different circuit boards.
  - C. All devices are assigned different addresses.
  - D. The proper logic level must be sent out on the MPU VMA and E lines.
  
3. Which of the following devices ensures that only one circuit will be enabled at a given time?
  - A. Buffers.
  - B. Addressable latches.
  - C. Decoders.
  - D. Line drivers.

Refer to Figure 10-31 for questions 4-6.

4. Which of the devices shown in Figure 10-31 is a noninverting 3-state buffer enabled by a logic 0?
  - A. The device shown in Figure 10-31A.
  - B. The device shown in Figure 10-31B.
  - C. The device shown in Figure 10-31C.
  - D. The device shown in Figure 10-31D.

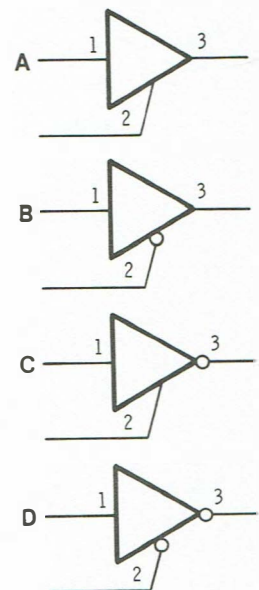


Figure 10-31  
Figure for questions 4 – 6.

5. In Figure 10-31D, in order to obtain a logic 1 at pin 3, which of the following must take place:
- A. There must be a logic 1 on pin 1 and a logic 0 on pin 2.
  - B. There must be a logic 0 on pin 1 and a logic 0 on pin 2.
  - C. There must be a logic 0 on pin 1 and a logic 1 on pin 2.
  - D. There must be a logic 1 on pin 1 and a logic 1 on pin 2.
6. Which of the following devices shown in Figure 10-31 is an inverting 3-state buffer, disabled by a logic 1 applied to pin 2?
- A. The device shown in Figure 10-31A.
  - B. The device shown in Figure 10-31B.
  - C. The device shown in Figure 10-31C.
  - D. The device shown in Figure 10-31D.
7. In order for the MPU to read data from the RAM, which of the following conditions must exist?
- A. The  $R/\overline{W}$  line must be high and VMA must be low.
  - B. The  $R/\overline{W}$  line must be low and VMA must be high.
  - C. The  $R/\overline{W}$  line must be low and VMA must be low.
  - D. The  $R/\overline{W}$  line must be high and VMA must be high.
8. In the ET-18, when does reset occur?
- A. When it is manually entered through the keyboard.
  - B. When the ET-18 ceases all operations.
  - C. When power is initially applied.
  - D. Both A and C are correct.

9. Referring back to the RAM storage cell shown in Figure 10-11, in order to store a binary 0 in the cell, which of the following conditions must be met.
- A. Logic 1 must be applied to the input line and a momentary logic 1 must be applied to the word select line.
  - B. Logic 0 must be applied to the input line and a momentary logic 1 must be applied to the word select line.
  - C. Logic 0 must be applied to the input line and a momentary logic 0 must be applied to the word select line.
  - D. Logic 1 must be applied to the input line and a momentary logic 0 must be applied to the word select line.
10. Referring to Figure 10-17 in the text and considering that both VMA and  $\overline{R\overline{W}}$  are high, an address of 000000111111101<sub>2</sub> would.
- A. Write data into RAM U4.
  - B. Write data into RAM U5.
  - C. Read data from RAM U4.
  - D. Read data from RAM U5.
11. Which of the following devices can be programmed by the user, but once the device is programmed, the contents of the device cannot be altered?
- A. PROM.
  - B. Mask-programmed ROM.
  - C. EPROM.
  - D. EEROM.
12. A ROM with 10 address and 8 data lines going to it could have a maximum memory capacity of:
- A. 1024<sub>10</sub> bits.
  - B. 2048<sub>10</sub> bits.
  - C. 1024<sub>10</sub> bytes.
  - D. 2048<sub>10</sub> bytes.

13. In most 6808 based microprocessor systems and the ET-18, which order addresses, high or low, are assigned to RAM and ROM?
- A. High order to RAM and low order to ROM.
  - B. Low order to RAM and high order to ROM.
  - C. Low order to both RAM and ROM.
  - D. High order to both RAM and ROM.
14. Referring to Figure 10-26 and 10-27 in the text, which of the following set of conditions must exist to output phoneme codes to the speech synthesizer?
- A. Address  $1100\ 0010\ 0100\ 0000_2$  must be on the address lines, VMA must be high,  $R/\overline{W}$  must be high, and E must be going from high to low.
  - B. Address  $1100\ 0010\ 0010\ 0000_2$  must be on the address lines, VMA must be high,  $R/\overline{W}$  must be low, and E must be going from low to high.
  - C. Address  $1100\ 0010\ 0100\ 0000_2$  must be on the address lines, VMA must be high,  $R/\overline{W}$  must be low, and E must be going from low to high.
  - D. Address  $1100\ 0010\ 0010\ 0000_2$  must be on the address lines, VMA must be high,  $R/\overline{W}$  must be low, and E must be going from low to high.
15. Referring to Figures 10-27 and 10-29 in the text, which of the following set of conditions must exist to input data from the sonar timing circuit.
- A. Address  $1100\ 0010\ 0010\ 0000_2$  must be on the address lines, VMA must be low, and  $R/\overline{W}$  must be low.
  - B. Address  $1100\ 0010\ 0100\ 0000_2$  must be on the address lines, VMA must be low, and  $R/\overline{W}$  must be high.
  - C. Address  $1100\ 0010\ 0110\ 0000_2$  must be on the address lines, VMA must be high, and  $R/\overline{W}$  must be high.
  - D. None of the above set of conditions causes data to be inputted from the sonar timing circuit.



## EXAMINATION ANSWERS

For your convenience, the page where the correct answer can be found is shown following the answer.

1. A — Random Access Memory. [10-6]
2. C — All devices are assigned different addresses. [10-7]
3. C — Decoders. [10-7]
4. B — The device shown in Figure 10-31B. [10-10]
5. B — There must be a logic 0 on pin 1 and a logic 0 on pin 2. [10-10]
6. D — The device shown in Figure 10-31D. [10-10]
7. D — The  $R/\overline{W}$  line must be high and VMA must be high. [10-14, 16]
8. D — Both A and C are correct. [10-14]
9. B — Logic 0 must be applied to the input line and a momentary logic 1 must be applied to the word select line. [10-25, 26]
10. C — Read data from U4. [10-38, 39]
11. A — PROM [10-44, 45]
12. C —  $1024_{10}$  bytes. [10-50]

13. B — Low order to RAM and high order to ROM. [10-50]
14. C — Address  $1100\ 0010\ 0100\ 0000_2$  must be on the address lines, VMA must be high,  $\overline{R/\overline{W}}$  must be low, and E must be going from low to high. [10-57, 58]
15. D — None of the above set of conditions causes data to be inputted from the sonar timing circuit. [10-60, 62]

*Unit 11*

**INDUSTRIAL ROBOTS AT WORK**

## CONTENTS

Introduction .....	11-3
Unit Objectives .....	11-4
Unit Activity Guide .....	11-5
Industrial Robot Classification .....	11-6
Application Considerations .....	11-23
Experiment 19 .....	11-34
Unit Examination .....	11-35
Examination Answers .....	11-38

## INTRODUCTION

Webster's Seventh New Collegiate Dictionary defines a robot as "a machine that looks like a human being and performs various complex acts (as walking and talking) of a human being." Webster's definition of a robot may be appropriate for a majority of the population, however, the Robot Institute of America (RIA) has a definition better suited to the needs of industry. The RIA definition is as follows: "A robot is a reprogrammable multifunctional manipulator designed to move material, parts, tools, or specialized devices, through variable programmed motions for the performance of a variety of tasks."

In this unit, you will learn how the RIA definition applies to industrial robots. Before we proceed with our studies of some specific industrial robot applications, we will take a few moments and look at the history of these dedicated devices.

Industrial robots first appeared on the scene in the early 1960's but saw only limited duty during their first decade of existence. During the 1970's, industrial robot applications showed a steady growth in numbers and in variety of uses. During the 1980's, however, the phenomenal growth rate expected of industrial robots is almost unpredictable. The following is what one robotics expert has to say regarding the use of electronically controlled machines:

"The human race is now poised on the brink of a new industrial revolution which will at least equal, if not exceed, the first industrial revolution in its impact on mankind. The first revolution was based on the substitution of mechanical energy for muscle power. The next industrial revolution will be based on the substitution of electronic computers for the human brain in the control of machines and industrial process." — Mr. James S. Albus, Director, Robotics Research Laboratory, National Bureau of Standards.

When studying the industrial robots in this unit, one important factor must be taken into consideration. Robots can never be any smarter than the person who gives them instructions.



## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. State the primary differences between limited sequence, point-to-point, and continuous path robots.
2. Explain the difference between “walk through” and “lead through” methods of teaching.
3. Determine which type of robot programming is best suited for a particular industrial task.
4. State the difference between command and error signals.
5. Explain the use of feedback devices in a servo controlled robot system.
6. Describe the following manipulator configurations:
  - \* Cylindrical coordinate
  - \* Cartesian coordinate
  - \* Polar coordinate
  - \* Revolute coordinate
7. Describe various types of end effectors and how they are used.
8. Explain why interfacing is so important in robot applications.
9. State the three most common wrist assembly axis of motion.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read "Industrial Robot Classification."	_____
<input type="checkbox"/> Answer Programmed Review questions 1-13.	_____
<input type="checkbox"/> Read "Application Considerations."	_____
<input type="checkbox"/> Answer Programmed Review questions 14-25.	_____
<input type="checkbox"/> Perform Experiment 19.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## INDUSTRIAL ROBOT CLASSIFICATION

In Unit 1, we categorized industrial robots according to their technology level: low, medium, and high. Classification of industrial robots is by no means an easy task — with hard, fast rules — since their capabilities range from very simple to extremely complex. The ideal situation would be to classify all robots by their job capabilities; however, due to their rapid evolution, a good deal of overlap would occur. While a simple pick-and-place robot may be quite capable of doing a specific task, a more sophisticated robot could possibly do the task even better. Conversely, a more sophisticated robot is sometimes an overkill.

Regardless of their classification, all robots contain two major systems. First, there is the mechanical system which includes the base, manipulator, and gripper. These are the most obvious parts of the robot since they can be readily seen. Second, there is the control system which is not so readily discernible but actually is the “brains” of the robot. The control system can be as simple as a series of adjustable mechanical stops or limit switches, or it can be a complex computer-controlled system that gives the robot a programmable memory.

For the purpose of this discussion we have divided industrial robots into three classifications: limited sequence, point-to-point, and continuous path. Of course, there are other classifications, but these are the most commonly used in industry.

### Limited Sequence Robots

The limited sequence robot, commonly referred to as a pick-and-place or nonservo robot, is the least sophisticated class of robot. This is not meant to imply that these robots do not have a future in the industrial process. Quite the contrary. There are many industrial applications that require a robot capable of performing only relatively simple tasks. To perform these tasks, the robot requires only a **limited** number of **sequential** actions. Therefore, it does not require the motion capabilities, thus the control, of the more sophisticated robot.

The limited sequence robot uses a system of mechanical end stops and limit switches to electromechanically control the manipulator movements. The electromechanical switching is achieved by using relays and rotary or stepping switches. The operational sequence is determined by physically configuring patchboards, rotating cams, or in the case of air-logic systems, connecting various air tubes. Using this type of control, only the end positions of each axis of manipulator motion can be specified and controlled. For example, the manipulator could be extended from point A to point B, but the path between these two points is not defined. Thus, the controls simply switch the drives on and off at the ends of travel with no intermediate stops.

Figure 11-1 is a block diagram of a limited sequence robot. The robot can be actuated hydraulically, pneumatically, electrically, or by a combination of these methods. Limited sequence robots are generally smaller, but tend to move faster than larger more sophisticated ones. Speeds of 30 - 60 inches per second in all axes of travel is quite common. The use of mechanical end stops and limit switches gives good positional accuracy, with repeatability of 0.010 inch being quite common.

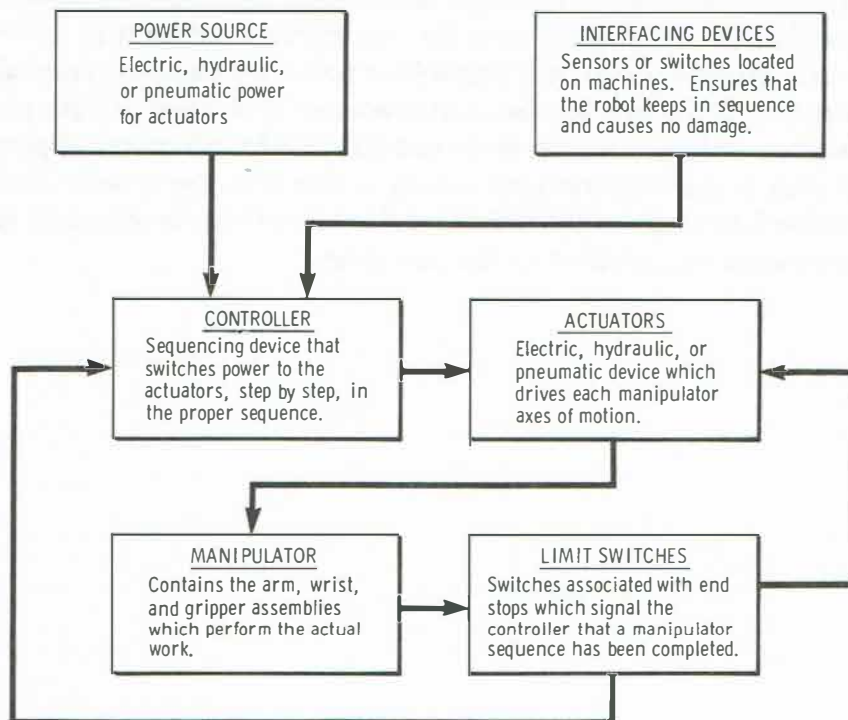


Figure 11-1

Limited sequence robots are lower in cost than the larger robots. Along with their lower initial cost, limited sequence robots are generally cheaper to operate because of their relatively low maintenance requirements. Because of the lower costs associated with these robots, they can be applied to many tasks where the more sophisticated robot would not be cost effective. They have been used successfully in a variety of applications such as die-casting, press loading and unloading, plastic molding, and parts handling operations.

Devices of this type have disadvantages other than the obvious control limitations. The number of motions the manipulator can perform is severely restricted by the limited number of mechanical end stops and limit switches that can be placed on the robot. Also, limited sequence robots **cannot be taught** to perform their task. Instead, to program them, you usually must physically set up all the end stops in their appropriate positions, and adjust the contacts in the controller so that all steps take place in the correct order. This type of programming is very tedious and time consuming when compared to programming the more sophisticated robots.

Unlike the robots we will discuss later, the limited sequence robot cannot provide any real control over the manipulator while it is actually in motion. In some cases, it is possible to provide a stopping point along a given axes of motion, but the limited amount of memory and the physical placement of switches severely restricts this for all practical purposes. Because of these restrictions, robots of this type are usually limited to three or four degrees of freedom, and such options as wrist and gripper movements are specified by the customer.



## BASIC OPERATION

The basic operation of a limited sequence robot is as follows:

1. When the desired preprogrammed task is initiated, the controller will switch power to the applicable manipulator actuating motor. If the actuating motor is electric, the controller will cause a relay to switch the current through to the motor. If the actuators are hydraulic or pneumatic, then appropriate solenoid valves are operated.
2. The manipulator motion, along a given axes of travel, generated by the actuator will continue until the manipulator is physically restrained by hitting an end stop. The physical shock of the stop is cushioned by some form of shock absorbing device.
3. The device is constructed so that the limit switch cuts off power to the actuator as soon as the end stop has been reached. At the same time, the limit switch also signals the controller that the particular movement is finished, and the next manipulator motion can be started. As you can see, there are only two positions at which the manipulator can come to rest, one at the beginning and one at the end of the programmed move.
4. The controller is then sequenced, or stepped, to the next point in the program. The controller could be a set of tabs placed on a drum or a set of contacts operated by a cam, either of which can be rotated a few degrees at a time by a stepping motor. Each time the controller receives a signal to drive, it steps to the next position. This action causes power to be switched on or off, which results in manipulator motion along a particular axes of travel.
5. This procedure is repeated, step by step, until the programmed task is completed. The robot is then ready to start another work cycle.

There is another important fact about limited sequence robots. All robot applications require that the robot be integrated with the system or machine it is serving. Even a simple robot operation such as unloading a machine cannot be completed successfully without interfacing with the machine it is serving and the surrounding work area.

Remember, unlike a human worker, a robot is deaf, dumb, and blind, so the robot must be made aware of the real world around it. Limit switches, or other sensing devices, on the machine and in the surrounding area are connected to the controller to provide control signals in addition to those obtained from the switches mounted on the robot itself. Thus, the robot's manipulator movements are carefully interfaced with the machine it is serving and the surrounding work area.

Interfacing prevents the robot from damaging the machine it is serving, damaging itself, or causing injury to surrounding equipment or personnel. Also, it enables the robot to carry out its assigned task not only in the correct sequence, but also at the appropriate time. Interfacing between the robot and the surrounding environment will be discussed in greater detail later in the unit.

One main disadvantage of the limited sequence robot is that it is difficult and time consuming to program, because of the **open-loop** control system and type of memory used. The electromechanical limit switches, end stops, interlocks, and controller tabs or contacts all have to be physically set. Not only is this arrangement time consuming, it also limits the number of specific steps the robot can perform; thereby, limiting the tasks it can be expected to perform.

## Programmable Robots With Point-to-Point Control

Closed-loop servo mechanisms are a much better method of controlling manipulator motion. Figure 11-2 illustrates such a control system in block diagram form. In this case, each manipulator joint is fitted with a feedback device that produces an electrical signal. This signal is referred to as an error signal, the value of which is proportional to manipulator position.

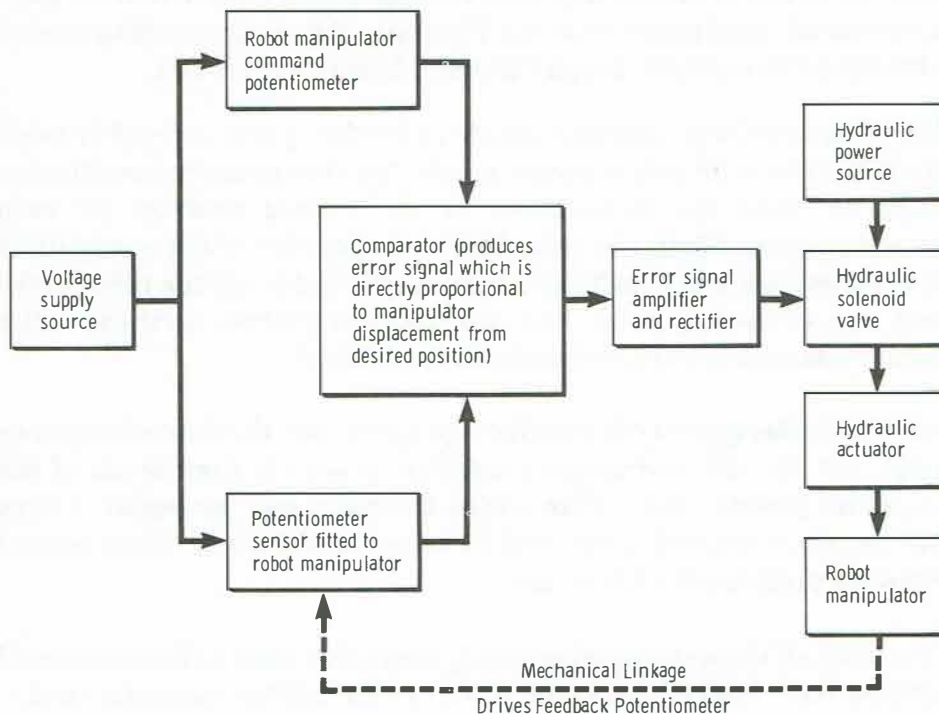


Figure 11-2

The system is designed so that the error signal decreases as the manipulator moves closer to the desired programmed position. When the error voltage has been reduced to zero, the manipulator is at the desired position. This is known as analog control, and requires a high degree of engineering skill in its design, to provide satisfactory positional accuracy without oscillation.

A positioning knob, added to the control panel, can vary the command signal for a particular axis of manipulator motion. The manipulator will then move as the knob is moved. If each manipulator axis of motion has its own positioning knob, an operator can remotely position the manipulator anywhere within its work envelope.

However, this device must have a controller with a memory unit before it can be considered a true robot. Once the controller is added, generally a microcomputer, a very versatile robot emerges. Now, the controller can store manipulator positioning data, corresponding to each desired step, and the total operational sequence. The controller memory is then used to drive the servo system through the desired programmed task.

The procedure for programming such a robot for a particular task is much easier than for a limited sequence robot. You simply use the positioning knobs to drive the manipulator to the desired position for each operational step. Next, you record the exact position of the manipulator in memory by simply pushing a button, before driving the robot to the next step of the sequence. You continue this process until the entire operational sequence has been placed in memory.

Not only is this type of robot easier to program than the limited sequence robot, but the microcomputer controller is able to digitize all of the command position data. This means the robot can remember a large number of sequential steps, and in many applications, store several complete programs for future use.

Even with all these capabilities, this type of robot must still be interfaced with the real world for the same reasons as the limited sequence robot.

### BASIC OPERATION USING POINT-TO-POINT CONTROL

The configuration shown in Figure 11-3 illustrates the operation of a robot using point-to-point control. In this example, the robot is required to pick up metal billets, (semifinished iron or steel ingots) from the end of a conveyor, and place them into a container fitted with 20 compartments. An optical sensor on the conveyor ensures that the conveyor stops as soon as the billet has been delivered to the pickup point. This prevents the parts from jamming up at the end of the conveyor and confusing the robot. In addition, guide walls are placed on the conveyor to ensure the robot can find the billet in the same place everytime, and that the billet is standing on one end.

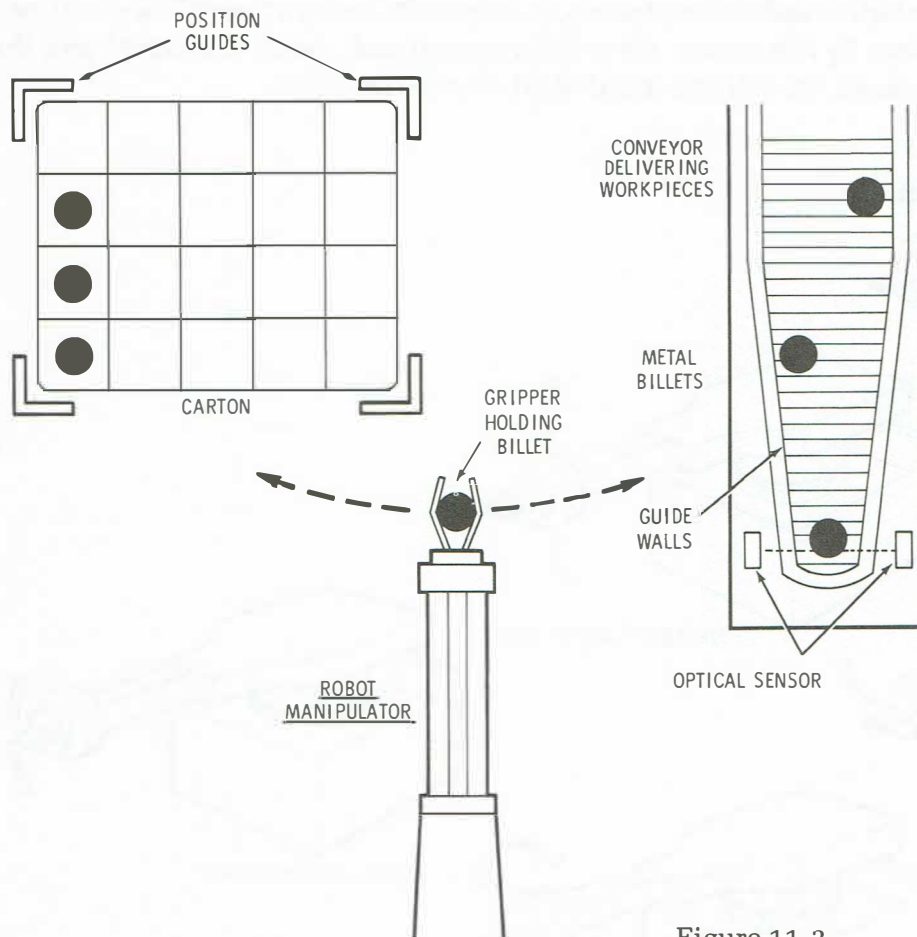


Figure 11-3



Container location is also very important. It is oriented on a table, using position guides, to guarantee that, once the carton is filled and removed, a new, empty carton will be placed in the exact location as the previous one.

The robot is going to be programmed to pick up the metal billets from the same position on the conveyor each time, which is a relatively easy task. However, it will also have to place the billets, one at a time and in a specific sequence, into 20 separate compartments in the carton. This requires a robot with far greater memory capabilities than the limited sequence device can provide.

Figure 11-4 depicts the robot to be used for this task. To program the robot for this task, the human operator will lead the robot through each step of the operation, one at a time, recording each step in the robot's memory. To make this task easier, the robot can be switched to a "teach" mode of operation. In this mode, the robot's operational speed is reduced and the operator has full command of all robot movements.

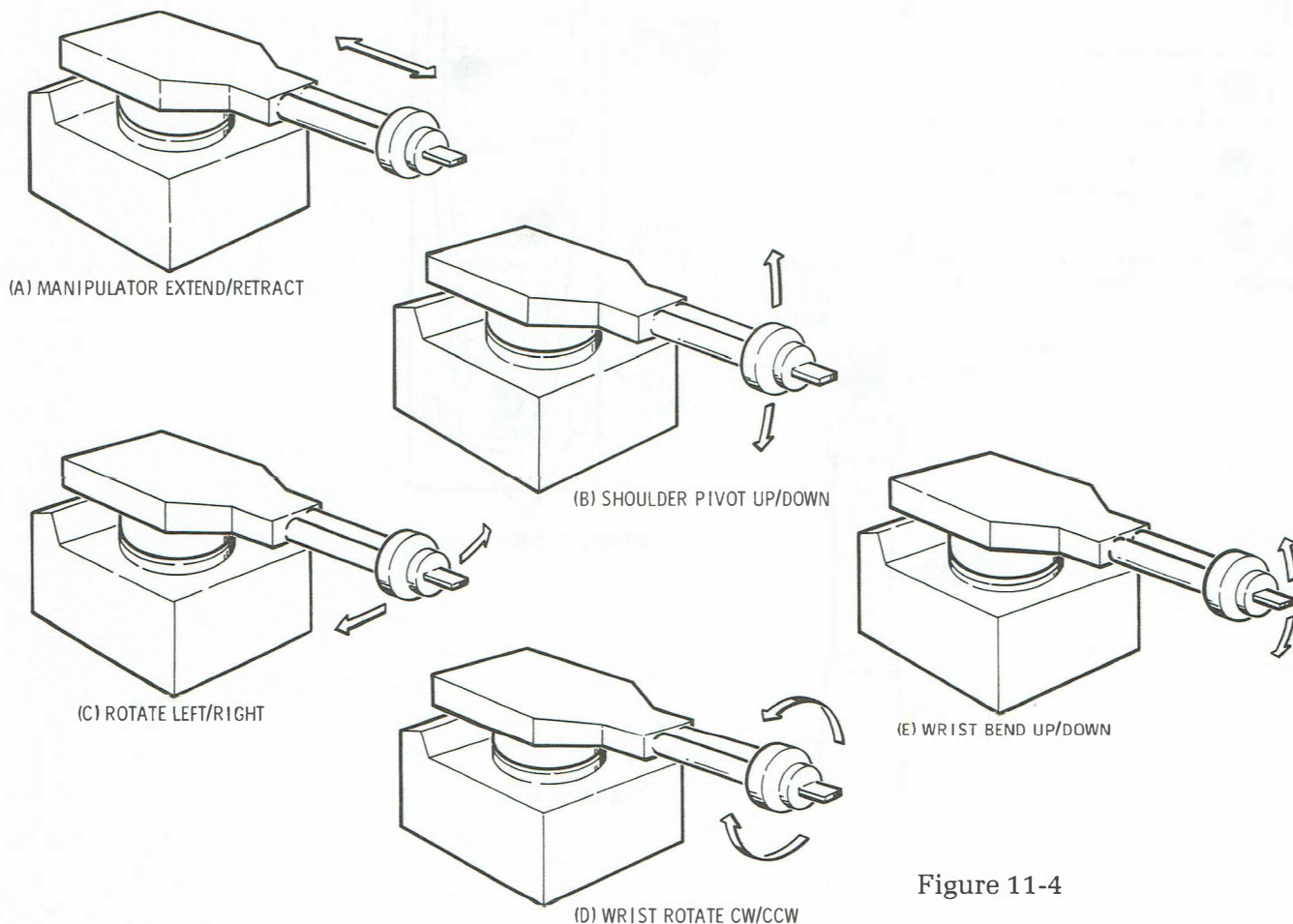


Figure 11-4

The operator is provided with a hand held “teaching pendant” similar to the one shown in Figure 11-5. This allows the operator to position himself in the best location to observe and control the robot’s operation during the teach mode. Now, by using the positional controls on the teaching pendant, the operator can guide the robot through each step to program the task. These controls are similar to the ones on the master control panel previously discussed, and perform the same function. This method of teaching is referred to as “lead-through.”

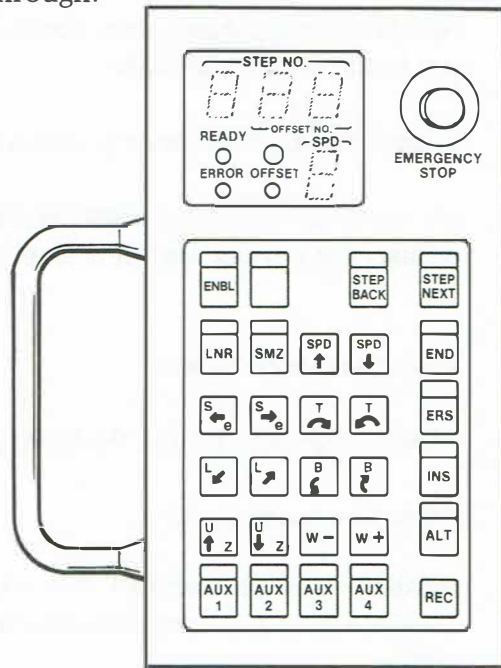


Figure 11-5

There are variations to this teaching method. In some instances, the robot manipulator is physically lead through the task using a device attached to the manipulator itself. This method is referred to as “walk-through” teaching.

In order for the robot to operate in such an environment, two interfacing switches are necessary. One is placed at the carton, to prevent the robot from trying to load billets before an empty carton has been put into place. The other is taken from the optical device that controls the conveyor. While it is important to stop the conveyor from depositing more than one billet at a time at the pickup point, it is also necessary to stop the robot when no billet is at the pickup point. Obviously, it would be rather strange to observe the robot going through all the motions with no billet to handle. Again, we see that the robot needs assistance to properly perform the task.

Now that we have arranged for the metal billet to be at the pickup point and in the correct position, it is time to program the robot for the task. Using the teaching pendant, and with the robot switched to the teach mode, the teaching/programming routine would be as follows:

1. With the gripper open, move the manipulator until the gripper is just above the metal billet to be picked up.
2. Rotate the gripper and wrist controls as necessary to align the gripper in the horizontal plane.
3. Record the manipulator's position by pressing the "record" button.
4. Lower the manipulator until the gripper surrounds the billet. "Fine adjust" the gripper until it is level and symmetrical around the billet.
5. Press the record button.
6. Close the gripper so that the billet is grasped firmly.
7. Press the record button.
8. Raise the manipulator until it is well clear of the conveyor, so that it does not hit any obstructions between the conveyor and the container.
9. Press the record button.
10. Rotate the robot and extend the manipulator until the billet is positioned above the first compartment in the carton.
11. Press the record button.
12. Lower the manipulator carefully until the billet is just above, but not touching, the first compartment in the carton. Ensure that the billet is in the vertical position, and if necessary, adjust the gripper and wrist controls to achieve this.

13. Press the record button.
14. Lower the manipulator until the billet rests in the bottom of the carton.
15. Press the record button.
16. Open the gripper.
17. Press the record button.
18. Raise the manipulator clear of all obstructions.
19. Press the record button.
20. Rotate the robot back to step number 1 and repeat the sequence until each compartment of the carton has been filled.
21. Replace the container with an empty one, switch on the conveyor, switch the robot to the operate mode, switch the robot to operate and watch the robot repeat the task.

If all interfacing switches have been correctly set, and the robot receives a steady supply of billets and empty cartons, the robot will repeat this routine in a regular, accurate, and untiring manner. During operation, the robot will stop briefly at each position the record button was pushed during the teaching routine. However, the robot will cease the routine if there is no billet to be picked up, if a full carton has not been picked up, or if the robot itself is switched off.

It is important to realize just how the robot now operates once it has been programmed. When it is commanded to move from one position to the next, this could involve two or more axes of motion of the manipulator. The only thing the robot has been taught, however, is the position of all the manipulator's axes at the start of the move and the new positions of the manipulator's axes when the move is completed. While making the move as fast as it can, and while moving all axes simultaneously to carry out the ordered command, there is no clear definition of the path the manipulator will follow. Thus, this type of robot is controlled "point-to-point," as the name implies. This is why it was necessary to record some intermediate robot positions in the program.

For example, if step 8 had not been recorded, the robot could very easily have hit an obstruction when moving between the conveyor and the carton. By the same token, steps 10 and 12 were recorded to ensure that the robot did not try to place the billet in the carton at an angle. It is important to remember that each major change of the robot's trajectory **must be recorded** when using point-to-point control.

Robots with point-to-point control, operating in a digital format, have virtually unlimited memory available. In the example just presented, the operator programmed in nine separate record points (steps 3, 5, 7, 9, 11, 13, 15, 17, and 19) for each billet to be handled. This amounts to 180 recorded steps for all 20 billets.

Point-to-point robots are more than capable of doing any task performed by the limited sequence robot. If sufficient memory is available, they can do even more sophisticated tasks such as spot welding, stacking, palletizing, etc.

## **Programmable Robots With Continuous Path Control**

There are many tasks in industry where it is necessary to not only control the starting and finishing points of each sequence, but also the trajectory traveled by the robot's manipulator between those points. Welding a car body is one example. Here, a robot is asked to hold a welding gun and move it along some complex contour while maintaining the correct speed to produce a good weld. The best method is "continuous path" control. As the name implies it provides continuous control of the robot's trajectory. Continuous path control is basically an extension of the point-to-point control just discussed, and like point-to-point control, it is used to control servo type robots.



With continuous path control, the operator can physically hold the manipulator, using a device similar to the one shown in Figure 11-6, and lead the robot through the motions it will later perform by itself. Here, you see the operator teaching a spray painting operation. (Do not confuse this lead-through action with the “lead-through” method of teaching previously discussed. Actually, this is considered to be the “walk-through” method of teaching.)

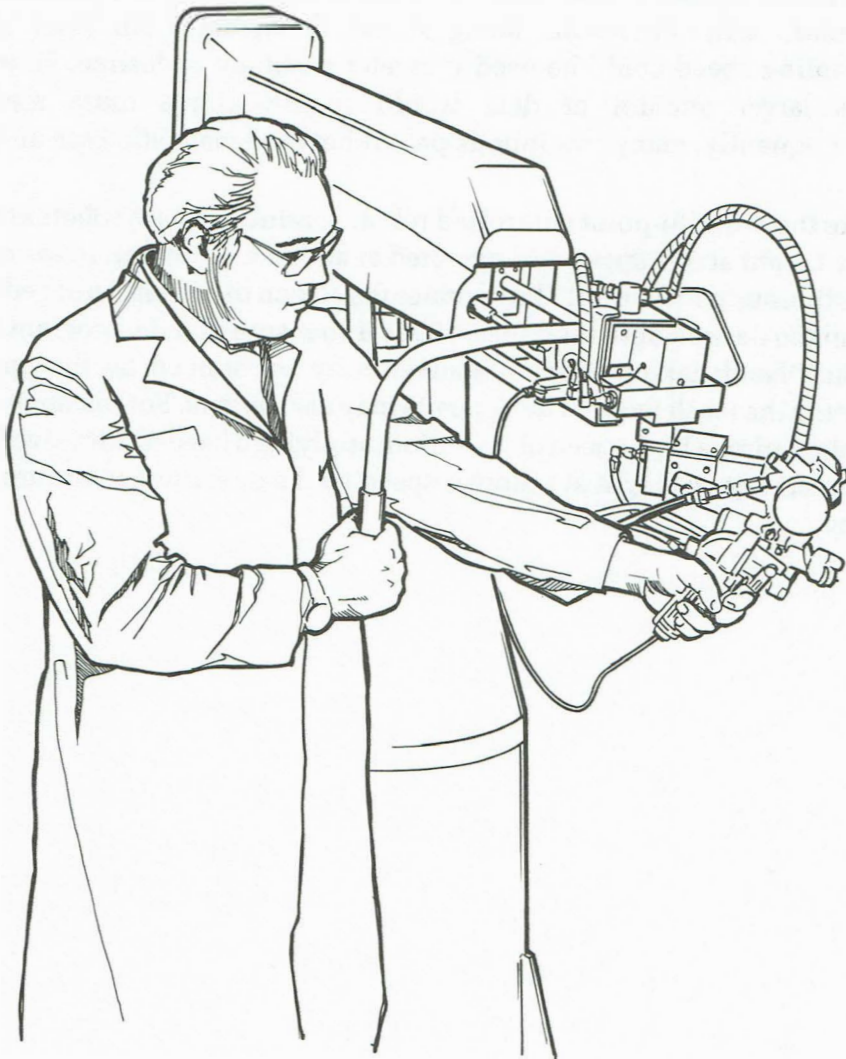


Figure 11-6

Since continuous path operation uses real time, the robot will perform the taught task at approximately the same speed as a human doing the same task. During the teaching process, the memory unit has to record all manipulator movements, continuously. This could be accomplished by giving the robot an internal timing system. For example, the robot could be synchronized with the 60 or 400 Hz main supply. Using this as a reference, the robot's movements would be sampled 60 or 400 times each second, with the results being stored in memory. An even higher sampling speed could be used if greater accuracy is desired. However, this larger amount of data would require much more memory. Consequently, many continuous path robots use magnetic tape units.

Like the point-to-point controlled robot, continuous path robots are usually taught at one speed and operated at another. However, in the case of continuous path control, the operator may teach the robot at a speed faster than the desired operating speed. This is to insure that the program is free from "handshaking effects" inadvertently introduced by the operator during the teach mode. The opposite may also be true. For instance, a task that requires a high speed of operation, applying a bead of adhesive for example, may be taught at a slower speed so the operator can ensure accuracy.

## Programmed Review

1. A limited sequence robot is an example of a \_\_\_\_\_ robot.  
(servo/nonservo)
2. (nonservo) The limited sequence robot uses a system of mechanical \_\_\_\_\_ and limit switches to electromechanically control manipulator movements.
3. (end stops) When compared to more complex robots, limited sequence robots generally have a \_\_\_\_\_ operating speed.  
(slower/faster)
4. (faster) Regarding limited sequence robots, each axis of motion can generally assume only \_\_\_\_\_ positions.
5. (two) Limited sequence robots \_\_\_\_\_ be taught to perform a specific task.  
(can/cannot)
6. (cannot) Limited sequence robots are generally \_\_\_\_\_ to program than the more sophisticated robots.  
(harder/easier)
7. (harder) When using closed-loop control, each joint of the manipulator contains a \_\_\_\_\_ device that produces an electric signal.
8. (feedback) The signal produced by the feedback device is referred to as an \_\_\_\_\_ signal.
9. (error) As the manipulator of a servo controlled robot moves closer to its command position, the error signal \_\_\_\_\_.  
(increases/decreases)

10. (decreases) In the walk-through mode of teaching, the robot's manipulator is \_\_\_\_\_ maneuvered through the assigned task.  
(automatically/manually)

11. (manually) Continuous path programming requires \_\_\_\_\_ controller memory than point-to-point programming.  
(more/less)

12. (more) Continuous path operation is accomplished in a \_\_\_\_\_ time environment.

13. (real) All robot applications require that the robot be \_\_\_\_\_ with the machine it is serving.

(interfaced)

## APPLICATION CONSIDERATIONS

Besides the method by which the robot is programmed, limited sequence, point-to-point, or continuous path, there are other factors that you must consider when selecting or using an industrial robot. These include items such as manipulator configuration, wrist action, end effector selection, workpiece placement, and interfacing with the task.

### Manipulator Configuration

An industrial robot must be able to reach the machine it is serving or the workpiece it is handling. The robot's sphere of influence, (work envelope) is based upon its ability to deliver the end effector to specific locations in order to carry out the assigned task. A variety of manipulator configurations have been used, and so far, robot manufacturers have selected one or more of the following:

- \*Cartesian coordinates
- \*Cylindrical coordinates
- \*Polar coordinates
- \*Revolute coordinates



As seen in Figure 11-7, each of these configurations offers a different shape to its respective sphere of influence. For different tasks, different manipulator configurations are required. For example, limited sequence operations could easily be accomplished using cylindrical coordinates, where only straight-line manipulator motion is required. At the other end of the spectrum, revolute coordinates would be required if the task dictated that the manipulator place the end effector in a bin to pick up a part. As shown, revolute coordinates most closely resemble the actions of the human arm.

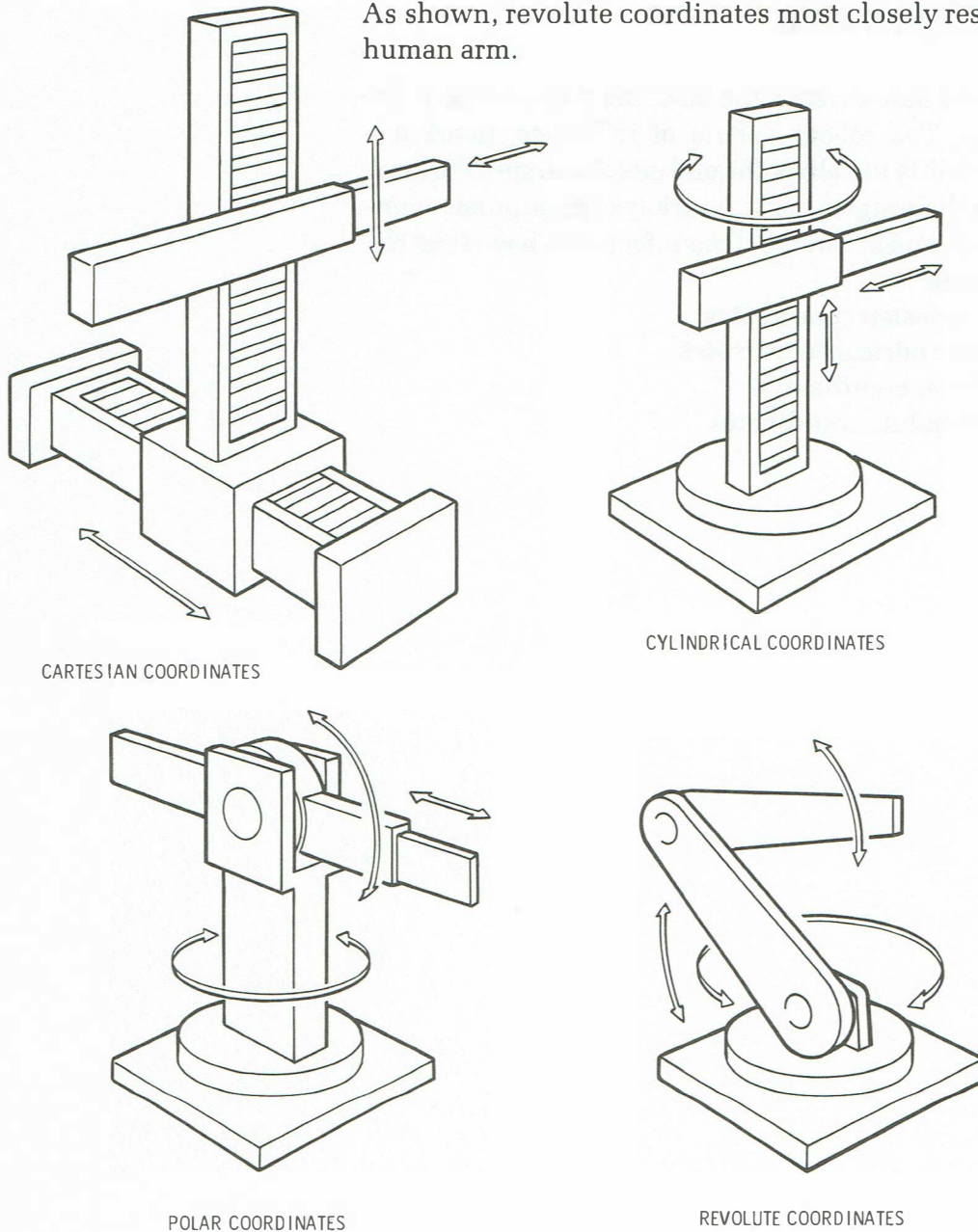


Figure 11-7

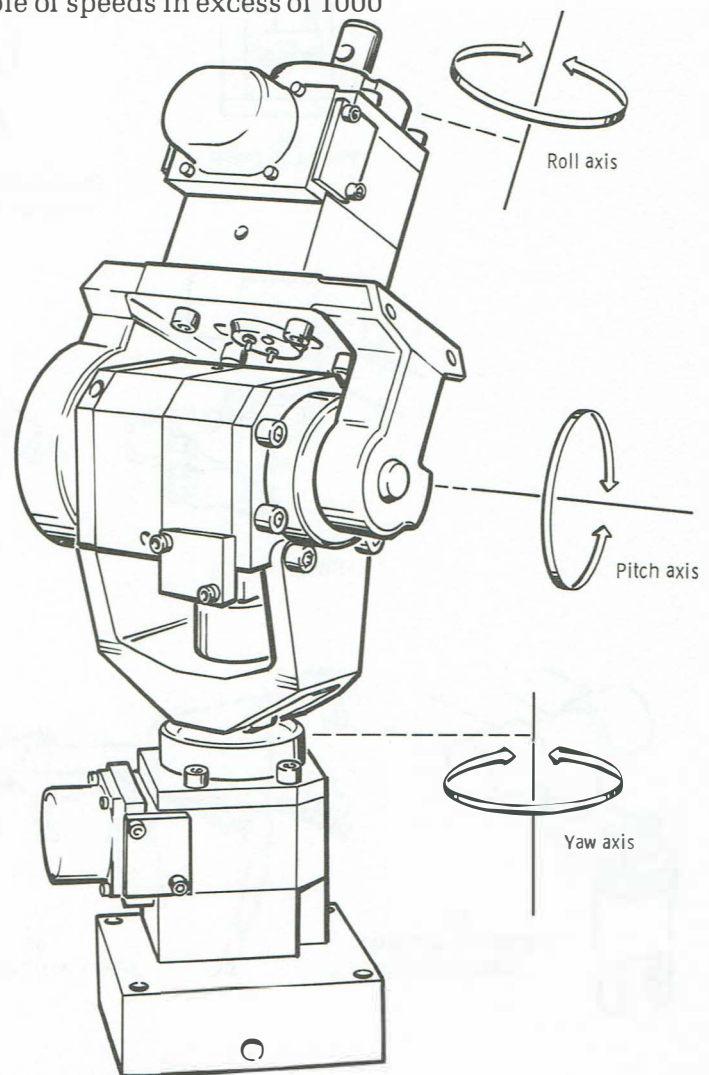
It should be noted that all of the manipulator coordinate systems require three degrees of freedom to position the wrist assembly in the work envelope. Then, three more degrees of wrist freedom are often required for universal orientation of the end effector.

## Wrist Assembly

Wrist action is used to provide the robot with an even greater amount of flexibility. In addition to the three degrees of freedom provided by the robots manipulator, the wrist can easily provide an extra three degrees of motion. These articulations are commonly referred to as yaw, pitch, and roll.

A state-of-the-art, electrohydraulic, 3-axis wrist assembly, developed by Moog Inc., is shown in Figure 11-8. The wrist rotates  $280^\circ$  in the yaw and roll axes,  $180^\circ$  in the pitch axis, and is capable of speeds in excess of 1000 degrees per second.

Figure 11-8



## End Effector

The end effector is the device that performs the actual work. An end effector can be a simple gripper type of mechanism or a special tool fitted to the robot's wrist assembly. Figure 11-9 shows some of the end effectors currently being used in industry.

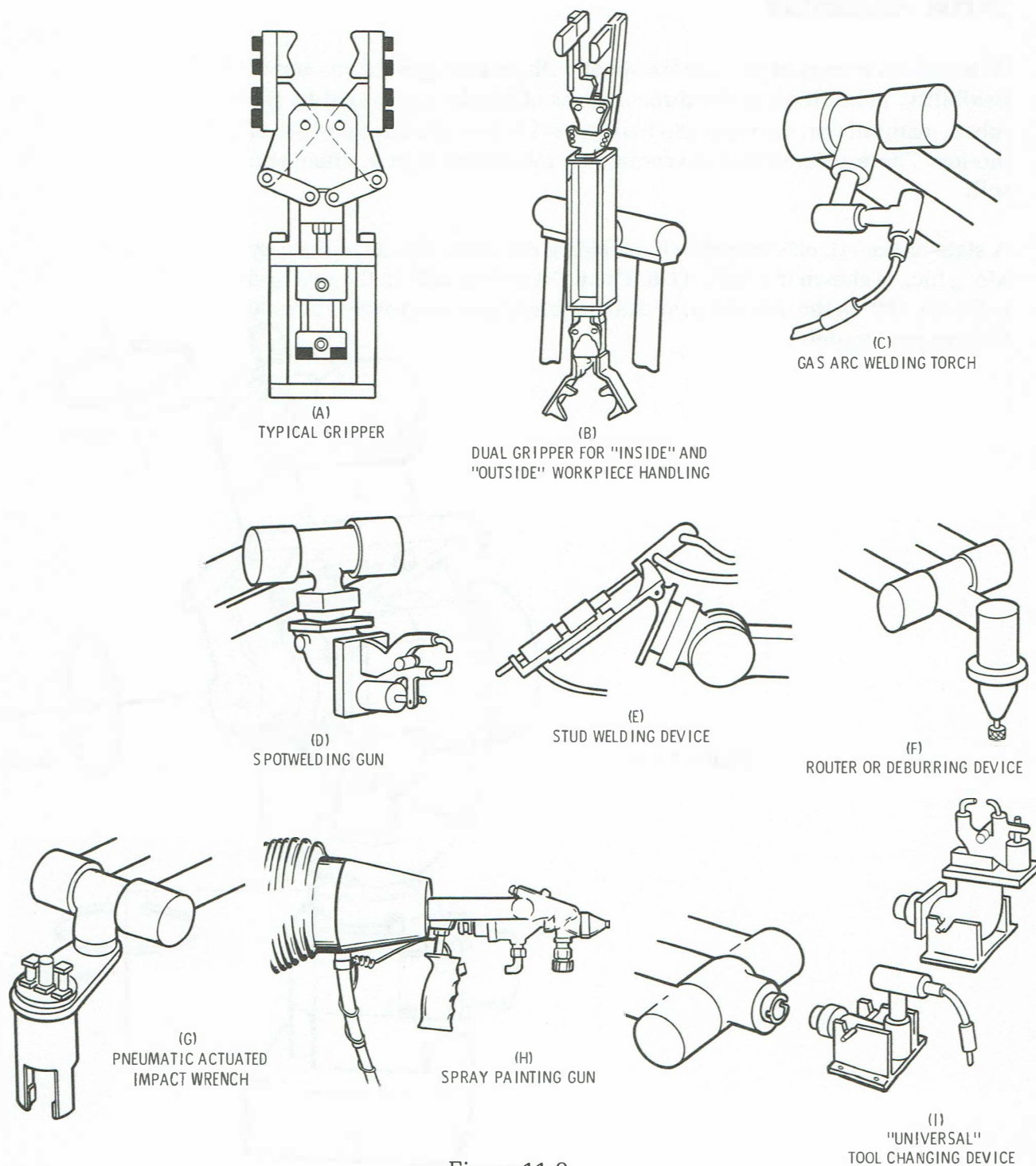


Figure 11-9

The device shown in Figure 11-9A is a typical two position gripper. That is, it is either normally open or normally closed. Many grippers of this type have interchangeable fingers that allow them to handle a variety of workpiece configurations. Figure 11-9B is a dual gripper that has the capability to grasp both the “inside diameter” and “outside diameter” of the workpiece. This allows the robot to unload a machine with one hand while holding an unprocessed workpiece in the other. This is especially advantageous, since it permits the robot to reload the machine without moving back to a pickup point to get the next part. These type grippers are attached to the wrist assembly and are actuated either pneumatically, electrically, or, if they have to handle heavy objects, hydraulically. There are some rare instances in which they are actuated mechanically.

Robots are not only used for material handling; they are also very effective tool handlers. In the first two examples, the robot carried the workpiece to the machine tool it was serving. In the following examples, the robot brings the tool to the workpiece. The inert gas welding torch, shown in Figure 11-9C, is very effective for curved welding runs, as well as short welds at different angles. Workpiece size is only limited by the robot's reach. Because of the continuous angle changes, this type of application would require continuous path programming.

The spotwelding gun, shown in Figure 11-9D, is used in line applications such as the welding of auto bodies. This application calls for stop-and-go rather than continuous operation; thus, point-to-point programming may be used. If several welds are required, as is the case in the auto industry, the work is divided between two or more robots. Positioning accuracy of the workpiece in this type of operation is extremely important.

In the stud welding device, shown in Figure 11-9E, welding studs are automatically fed to the gun from an overhead feeder. Again, workpiece positioning must be exact.

The end effector, shown in Figure 11-9F, is used to remove rough edges from a workpiece. It could just as easily be used for polishing or sanding operations. The workpiece would have to be held in a jig and, because the robot is required to perform a continuous motion, continuous path control is required.



The pneumatic-actuated end effector, shown in Figure 11-9G, is used to tighten nuts during an automobile assembly operation. The use of mechanical guides will greatly increase the locating accuracy of this device and help shorten positioning time. This type of end effector could also be used for drilling operations. Because the contour of the workpiece does not have to be followed, point-to-point control can be used.

The spray painting gun, shown in Figure 11-9H, can be used not only for painting operations but also to apply other finishes as well. Note the “walk-through” teaching pendant attached to the device. Because finishing operations require many complex, continuous moves, continuous path control is used.

The “universal” tool-changing device, shown in Figure 11-9I, allows a single robot to perform more than one operation. The tools can be of different types, permitting multiple operations on the same workpiece. With an automatic tool-changing operation programmed into the robot’s memory, the robot simply lowers the tool into a special fixture. The robot then rotates its wrist to unsnap the tool; this procedure is then reversed to pickup another tool and complete the assigned task.

There are many variations to the devices just discussed and many end effectors are designed to perform a specific task. For example, a vacuum device could be attached to the manipulator to handle flat workpieces, even glass. Consequently, it is the end effector of today’s industrial robot that is one of the most limiting factors in the development of the universal robot. This is mainly due to the lack of end effector programmability. However, extensive research and development is being conducted to produce an end effector that can handle a wide assortment of special tools.



## Workpiece Placement

Quite often, a robot does not require six degrees of freedom in order to accomplish a given task. We know that the manipulator positions the wrist assembly in such a manner as to orient its end effector as demanded by workpiece placement. However, in many instances, proper presentation of the workpiece or work area layout can alleviate the requirement for a full six degrees of freedom.

Actually, five degrees of freedom would be adequate if the workpieces were arranged so as to reduce part manipulation requirements. An example would be workpieces that were all located parallel to one axis of a cartesian coordinate robot or on a radius of base rotation for cylindrical, polar, or revolute coordinate robots. Quite simply, the better the workpiece can be presented to the robot, the better the chance a robot will have in succeeding in its assigned task.

There are many devices dedicated solely to the placement of workpieces; however, it is not the intention of this course to discuss these devices. This would be the responsibility of the industrial engineer who designs and installs the overall system.

## Interfacing to the Task

As stated earlier, all robot applications require some form of interfacing with the outside world. In actuality, interfacing fulfills four basic requirements.

1. It protects the robot from the machine it is serving. For example, an interlock on a shearing machine would not allow the machine to cycle until it received a signal from the robot that it was clear of the machine.
2. It protects the machine from the robot. An example of this would be a stamping machine requiring a workpiece, and correct placement of the workpiece, before the stamping cycle could begin. If the robot failed to deliver, or incorrectly placed the workpiece in the machine, an interlock in the machine would not allow the machine to cycle, thus preventing damage to the machine.

3. Interfacing ensures that the robot is aware of its surrounding environment. For example, an application requiring a robot to unload a furnace when the furnace has reached a desired temperature. In this case, a temperature sensor in the furnace would inform the robot that the correct temperature has been reached and the part is now ready for unloading.
4. Last, and most important, interfacing protects human beings from injury by the robot. This could easily be accomplished by surrounding the robot's work area with a fence. Any access to the work area, through an interlocked gate, would signal the robot to cease operation immediately. Of course, there would have to be provisions made to allow an authorized operator to service the robot.

### INTERFACING EXAMPLES

Interfacing the robot and related equipment involves transmitting data in several directions. A common robot application, spray painting, will illustrate the extent to which interfacing may be required. The spray painting operation shown in Figure 11-10 requires that both left and right side automobile car doors be painted one of four different colors. The operation is as follows.

As the car door enters the spray booth, it is determined, by an external sensor (A), whether it is a left or right door. This information is sent to the robot's controller (B), via the auxilliary control unit (C), which controls all interfacing devices and also acts as a signal conditioning unit. The controller selects the proper program, for either a right or left door, and sends this information to the robot.

Since the robot can apply one of four different colors, through the use of a multi-head spray gun, the preprogrammed controller tells the spray painting control unit (D) which color to supply to the robot.

When the door is in place in the spray booth, another sensor (E), possibly an optical sensor, signals the robot, via the auxilliary control unit and controller, to commence operation.

When the robot has manipulated the spray gun to its first programmed position, the controller instructs the spray painting control unit to begin dispensing paint. At the same instant the spray painting control unit begins dispensing paint, it signals the controller, which in turn instructs the robot, to begin the programmed operation.

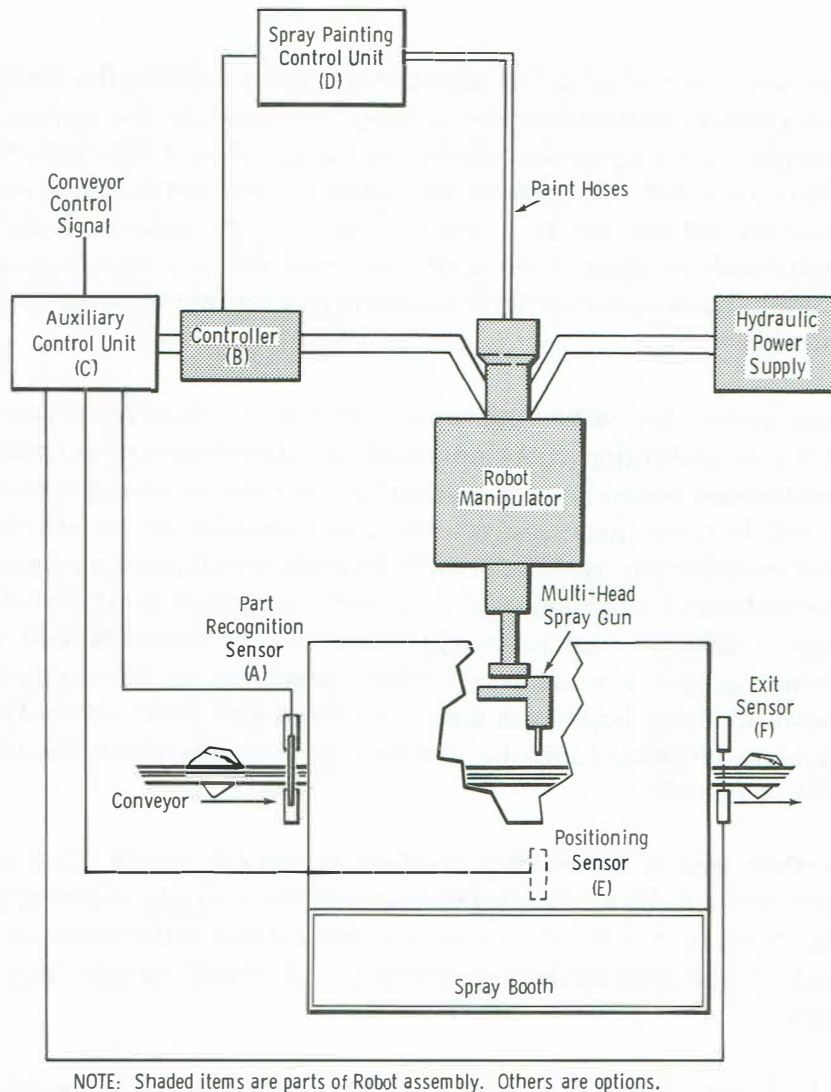


Figure 11-10

When the robot has completed the spraying operation, it signals the spray painting control unit, via the controller, to cease dispensing paint. The spray painting control unit in turn informs the controller it is no longer dispensing paint. The controller now instructs the robot to move out of the way.

Once the robot is out of the way, the controller signals the conveyor, via the auxiliary control unit, to move the door out of the spray booth. Another external sensor (F) signals the controller, via the auxiliary control unit, when the door is clear of the area. The controller uses this same signal to reset the robot, as another door is about to enter the work area. The robot is now ready for another left or right car door. This operation will continue until the robot is given a command to cease operation. As you can see, this operation requires more than just a robot.

Similarly, branching instructions can be initiated during the execution of a program by transmitting the appropriate signal to the controller. For example, in the operation shown in Figure 11-3, if the container into which the robot was placing the metal billets was full and an empty container not yet put in place, a signal may be used to interrupt the operational program. This interrupt could cause a branch to another program that would instruct the robot to an alternate container to dispose of the billet.

In operations where moving conveyors or workpieces are involved, line tracking capabilities (robot operation synchronized to the speed of the line) become necessary. Interfacing between the robot and the line is required. In these instances, resolvers, tachometers, or encoder feedback devices, driven by the line, provide a continuous flow of workpiece position and speed information to the robot's controller. By this method, the robot is able to maintain synchronization with the workpiece while it moves along the line. Thus, if the line speeds up, the robot speeds up its operation; if the line slows down, the robot also slows down. However, there are practical limits to how fast, or even how slow, the robot can work efficiently.

Another area of interfacing involves sensory feedback. One common application is the unstacking of material, for example, sheets of glass. In this application, a tactile sensor, a sensor related to the human feeling of touch, is mounted on the vacuum tool with which the robot handles the glass.

The robot is programmed point-to-point to advance its manipulator in the direction of the stack, with the last programmed stopping point a little beyond the last piece of glass in the stack. If there is glass in the stack, the sensor contacts the glass and signals the robot that it has contacted the workpiece. This signal interrupts the program, stops the advancement of the manipulator, and activates the vacuum tool. Once the glass has been gripped, a signal is sent to the robot to continue with the operational program, removing the glass from the stack and placing it in the appropriate area.

However, if there isn't any glass in the stack, the robot will reach the preprogrammed stopping point and sense the lack of glass. At this point, the robot can activate an alarm to inform the operator that the stack is empty.

Other types of sensors such as proximity detectors and vision systems may also be applied to the robot in a similar manner. As you can see, the addition of interfacing has greatly increased our robot's intelligence.

## Programmed Review

- |     |  |
|-----|--|
| 14. | In order for a robot to position the wrist assembly within the work envelope, the manipulator must generally have _____ degrees of freedom.                      |
| 15. | (three) Universal orientation of the end effector requires at least _____ degrees of total manipulator and wrist freedom.<br>(six/eight)                         |
| 16. | (six) Where only straight-line motion is required, the robot manipulator could be programmed most efficiently using _____ coordinates.<br>(cylindrical/revolute) |
| 17. | (cylindrical) The _____ is a device that performs the actual work.   |
| 18. | (end effector) Normally, a typical gripping device has _____ positions it can assume.  |
| 19. | (two) A gripper type end effector _____ be mechanically actuated.<br>(can/cannot)  |



20.	(can) A spotwelding application would most likely use a _____ controlled program.
21.	(point-to-point) It _____ possible for more than one robot to work on the same workpiece at the same time. (is/is not)
22.	(is) Some end effectors use _____ to increase locating accuracy and shorten positioning time.
23.	(mechanical guides) A _____ tool-changing device allows a single robot to perform more than one operation on a workpiece.
24.	(universal) In robotized operations, workpiece placement _____ of prime importance. (is/is not)
25.	(is) Proper workpiece orientation _____ decrease the total number of degrees of freedom required to perform a specific task. (can/cannot)
	(can)

## EXPERIMENT

Perform Experiment 19. After you finish the experiment, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

The following multiple choice examination is designed to test your understanding of the material presented in this unit. Read each question and all four answers. Select the answer you feel is most correct. When you have completed the examination, compare your answers with the correct answers that appear after the exam.

1. Which of the following operations would require the greatest amount of time to program?
  - A. An operation requiring limited sequence control.
  - B. An operation requiring point-to-point control.
  - C. An operation requiring continuous path control.
  - D. An operation requiring controlled path control.
2. Which of the following industrial tasks could be best accomplished using continuous path control?
  - A. Spot welding an automobile body.
  - B. Transferring parts from one location to another.
  - C. Applying an enamel finish to a dishwasher.
  - D. Tightening a nut on an automobile body.
3. One primary disadvantage of the limited sequence robot is:
  - A. Its slow operating speed.
  - B. Its high cost.
  - C. Its tedious and time consuming programming requirements.
  - D. Its high maintenance requirements.
4. A limited sequence robot ceases manipulator motion along a given axes of travel when:
  - A. It contacts a limit switch.
  - B. It physically contacts an end stop.
  - C. It receives a signal from the controller.
  - D. Power to the actuator is cut off.
5. Which of the following types of robot would normally use an open-loop control system?
  - A. Point-to-point controlled robot.
  - B. Controlled path controlled robot.
  - C. Continuous path controlled robot.
  - D. Limited sequence robot.

6. Which of the following types of control requires the greatest amount of memory?
  - A. Continuous path.
  - B. Point-to-point.
  - C. Pick-and-place.
  - D. Limited sequence.
7. Which of the following best describes the “walk-through” method of teaching?
  - A. Manipulator end stops and limit switches are physically set.
  - B. The manipulator is physically lead through the task.
  - C. The manipulator is taught using a portable, hand-held teaching pendant.
  - D. The manipulator is guided through the program using a preprogrammed routine.
8. One main advantage to real time operation is:
  - A. It is easier to program.
  - B. It requires less controller memory than other operations.
  - C. It more closely resembles human motions and speed.
  - D. It is taught at the same speed as it is actually performed.
9. Which of the following manipulator configurations would best be suited for reaching into an overhead bin to grasp a workpiece?
  - A. Cylindrical coordinate.
  - B. Revolute coordinate.
  - C. Polar coordinate.
  - D. Cartesian coordinate.
10. Which of the following is **not** normally considered to be a wrist assembly degree of freedom?
  - A. Pitch.
  - B. Yaw.
  - C. Traverse.
  - D. Roll.
11. Which of the following types of end effector actuation would best handle heavy workpieces?
  - A. Hydraulic.
  - B. Electrical.
  - C. Pneumatic.
  - D. Mechanical.

12. The most efficient method of controlling three robots assigned to an automobile spotwelding operation would be:
  - A. Continuous path.
  - B. Point-to-point.
  - C. Controlled path.
  - D. Limited sequence.
13. The factor most limiting the development of a universal robot is:
  - A. Manipulator programmability.
  - B. Wrist programmability.
  - C. End effector programmability.
  - D. Interfacing capabilities.
14. Which of the following interfacing configurations is the most important?
  - A. Protecting the robot from the machine it serves.
  - B. Protecting the machine the robot is serving.
  - C. Making the robot aware of its surrounding environment.
  - D. Protecting a human from injury by the robot.
15. In a typical robotic line-tracking operation, the robot will:
  - A. Slow its operation to match line speed.
  - B. Speed up its operation to match line speed.
  - C. Cease operation if there is any change in line speed.
  - D. Both A and B are correct, within limits.

## EXAMINATION ANSWERS

For your convenience, the page where the correct answer can be found is shown following the answer.

1. A — An operation requiring limited sequence control. [11-8]
2. C — Applying an enamel finish to a dishwasher. [11-18]
3. C — Its tedious and time-consuming programming requirements. [11-8]
4. B — It physically contacts an end stop. [11-9]
5. D — Limited sequence robot. [11-10]
6. A — Continuous path. [11-20]
7. B — The manipulator is physically lead through the task. [11-19]
8. C — It more closely resembles human motions and speed. [11-20]
9. B — Revolute coordinate. [11-24]
10. C — Traverse. [11-25]
11. A — Hydraulic. [11-27]
12. B — Point-to-point. [11-27]
13. C — End effector programmability. [11-28]
14. D — Protecting a human from injury by the robot. [11-30]
15. D — Both A and B are correct, within limits. [11-32]



## *Unit 12*

# Experiments

## CONTENTS

Introduction .....	12-4
Tools and Equipment .....	12-5
Format For The Experiments .....	12-6
Experiment 1: Robot Familiarization—Platform Mobility .....	12-7
Experiment 2: Robot Familiarization—Manipulator Axes of Motion .....	12-15
Experiment 3: On Board Logic Test Probe .....	12-21
Experiment 4: Manual Control of a DC Motor .....	12-27
Experiment 5: Straight Line Programs .....	12-39
Experiment 6: Arithmetic and Logic Instructions .....	12-53
Experiment 7: Program Branches .....	12-65
Experiment 8: Additional Instructions .....	12-101
Experiment 9: New Addressing Modes .....	12-125
Experiment 10: Arithmetic Operations .....	12-135
Experiment 11: Stack Operations .....	12-147
Experiment 12: Subroutines .....	12-157
Experiment 13: Sensors .....	12-179
Experiment 14: “Open-Loop” Control of a DC Stepping Motor .....	12-189
Experiment 15: Robot Voice Routine .....	12-195
Experiment 16: Robot Language—Motors .....	12-207

Experiment 17: Robot Language—Sensors and Sound . . . . . 12-229

Experiment 18: Cassette and Teaching Pendant Interface . . . 12-249

Experiment 19: Modifying a Taught Program . . . . . 12-267

## INTRODUCTION

This Unit contains 19 experiments that are to be run on your ET-18 Robot Trainer. At the end of Units 1 through 11, you will be instructed to perform one or more of these experiments.

The experiments have been carefully designed to reinforce your understanding of the material presented in the text by demonstrating concepts, principles, and basic applications.

The early programming experiments in this Manual are extremely simple. The later programming experiments become more complex, but you will be able to accomplish them as you become familiar with the instruction set and programming techniques. Before you finish this course, you will be writing and modifying programs that will enable you to fully utilize the many features of the ET-18 Robot Trainer. Many capabilities of the ET-18 Robot Trainer closely resemble the actual operational characteristics of Industrial Robots.

When you complete an experiment, return to the "Unit Activity Guide" of the unit that directed you to the experiment. This is important, because you will be jumping from one point to another quite frequently.

**NOTE:** If you have not already done so, we recommend that you become familiar with the ET-18 Robot Trainer by studying the Users Manual and Technical Manual included with your Robot Trainer. By studying these manuals, you will become familiar with the switches, circuit board placement, and many user routines called out in the experiments.

## Tools and Equipment

All of the components required to perform these experiments are provided with the course. However, in order to realize the full learning benefits of these experiments, you should have, or have access to, the following equipment:

- The ET-18 Robot Trainer with all options.
- A VOM, VTVM, or Digital Multimeter. You will need a general-purpose meter that is capable of measuring DC voltage, DC current, and resistance. Use a digital meter, if possible, as the resolution of the measurements will be much better.

Only four tools are necessary for all of the experiments contained in this course. These tools are:

- Long-nose pliers.
- Wire cutters.
- A wire stripping tool.
- A soldering iron.

You will use the pliers to straighten the bent ends of hookup wire, as well as to straighten or bend component leads so that they can be conveniently inserted into the experimental board.

You will need the wire cutters to cut the hookup wire to size, and the wire strippers to remove about 1/4" of insulation from each end.

If you have to buy these tools, check into the multi-function types. These may combine pliers/cutters, cutters/strippers, or pliers/cutters/strippers in a single tool. Many times, the cost of these tools will be substantially less than that of three separate tools.

You will need the soldering iron to construct an Auxiliary Test Cable. A low wattage, pencil-type iron is recommended.



## Format For The Experiments

The instructions for each experiment are presented in the following format:

**Objectives**—The objectives state what you will have learned once you complete the experiment. You should keep these objectives in mind as you conduct the experiment.

**Introduction**—The material presented under this heading states the purpose of the experiment. It also serves to refresh your memory of the material covered in the previous unit.

**Material Required**—This section will provide you with a list of components and other material you will need to complete the experiment. Set these parts aside at the beginning of the experiment, so you will have fewer distractive interruptions during circuit construction.

**Procedure**—A series of sequential steps which describe the instructions for setting up portions of the experiment. Questions may also be included at different points of the section.

**Discussion**—Each experiment will be discussed to review and highlight the important points you should have observed during the step-by-step procedure. Some of the more lengthy experiments will have more than one discussion, to ensure you have learned a specific point before starting the next procedure.

# Experiment 1

## *Robot Familiarization— Platform Mobility*

**OBJECTIVES:**

*To familiarize you with the terminology associated with Robots.*

*To familiarize you with the basic platform mobility capabilities of your Robot.*

*To acquaint you with keyboard entry of Robot commands.*

**Introduction**

In Unit 1, you were introduced to the terminology associated with industrial Robots. You also became familiar with three classifications of industrial Robots—low, medium, and high technology—along with their related axes of motion, or degrees of freedom.

To enhance your understanding of Robot terminology and the platform motion associated with Robots, you will enter a program into the ET-18 Robot Trainer which will allow it to act as your instructor. At the same time, you will become familiar with entering Robot commands through the microprocessor keyboard.

When you use the Trainer, carefully follow all of the operating instructions. The Robot's microprocessor controller can only perform properly if it is programmed properly. However, you do not need programming experience at this time; just follow the instructions provided in the experiment. Don't worry about what you are entering.

Once you begin programming, leave HERO's power on! Turning the Robot off, for even an instant, erases its programmable memory.

The ET-18 Robot Trainer User's and Technical manuals contain a great amount of useful information. If you have not already familiarized yourself with these manuals, you should do so at this time, before proceeding with the experiment.

**Material Required**

ET-18 Robot Trainer

## Procedure

1. Ensure that the Trainer batteries have a full charge.
2. Remove the battery charging and teaching pendant cables from the Trainer.
3. Clear an area approximately 6ft by 6ft on the floor to be used as an experimental area.

**WARNING: DO NOT PERFORM THIS OR ANY OTHER EXPERIMENTS WITH THE ET-18 ROBOT TRAINER ON A TABLE OR OTHER ELEVATED PLATFORM.**

4. Position the Trainer as shown in Figure E1-1.

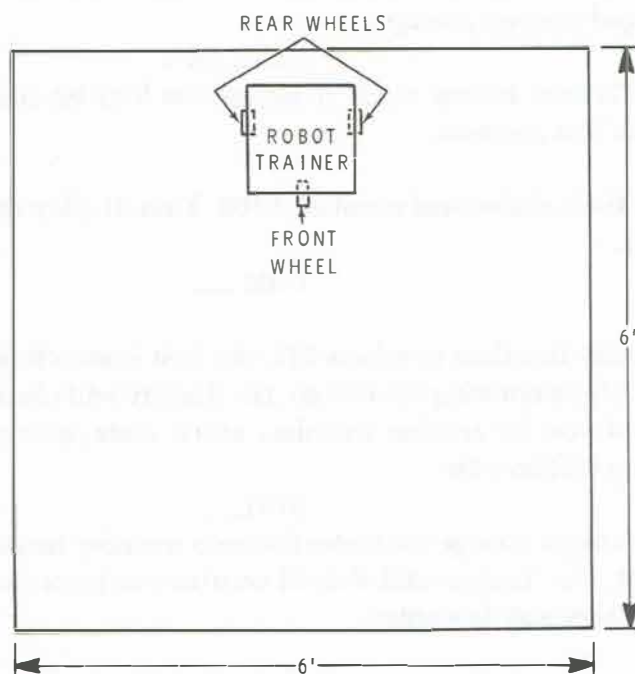


Figure E1-1  
Robot Trainer position.

5. Turn on the Trainer. The Robot's display should show: **HEro1.0** and its voice should say "ready". The Trainer is now in the executive mode awaiting your instructions. After approximately 10 seconds, the display will change to a moving dash. The Trainer is still in the executive mode, the change in display simply indicates a power conserving operation.
6. Using the Trainer's keyboard, perform an initialization procedure by depressing the 3 key followed by the 1 key. Your Trainer will indicate initialization completion both by display and voice. The entire initialization process takes approximately 90 seconds to complete.
7. Enter the Repeat Program Mode by depressing the 1 key. Once in the Repeat Mode, the display will indicate:  
1. \_\_\_\_\_
8. Press the **AUTO** (automatic) key. You will note that the display has changed to a new prompt:  
\_\_\_\_\_ Ad.  
Your Trainer is now ready to accept the four bit starting address of your first program.
9. Enter the hexadecimal numbers 0100. Your display should now indicate:  
0100 \_\_\_\_\_
10. Enter the inst/data numbers FD, the first instruction of this program. Upon entering the D digit, the display will clear and then re-prompt you for another inst/data entry. Note, however, the new prompt will now be:  
0101 \_\_\_\_\_  
This prompt change indicates the auto memory incrementing feature of your Trainer and should continue to increment after each successive inst/data entry.
11. Beginning with the second program line, enter the remaining inst/data numbers listed in Figure E1-2, the "Horizontal-Axis" program. After each inst/data pair entry, the address prompt will increment to the value shown for the next inst/data pair.

**Note:** Do not attempt to correct any errors you make during data entry, simply continue programming. Any errors will be located and corrected when you EXAMine your program later.



ADDRESS	INST/ DATA	ADDRESS	INST/ DATA	ADDRESS	INST/ DATA
0100	FD	0120	E8	0140	CC
0101	00	0121	91	0141	10
0102	00	*0122	3A	0142	00
0103	4D	0123	FD	0143	C3
0104	00	0124	00	0144	E8
0105	00	0125	00	0145	01
0106	62	0126	4D	*0146	3A
0107	49	0127	00	0147	FD
0108	C3	0128	00	0148	00
0109	10	0129	62	0149	00
010A	06	012A	49	014A	4D
010B	8F	012B	CC	014B	00
010C	00	012C	14	014C	00
010D	20	012D	20	014D	62
010E	C3	012E	C3	014E	49
010F	14	012F	E8	014F	CC
0110	0C	0130	86	0150	14
*0111	3A	*0131	3A	0151	23
0112	C3	0132	FD	0152	C3
0113	10	0133	00	0153	E8
0114	7E	0134	00	0154	00
0115	8F	0135	4D	*0155	3A
0116	00	0136	00	0156	FD
0117	20	0137	00	0157	00
0118	C3	0138	62	0158	00
0119	14	0139	49	0159	4D
011A	8A	013A	CC	015A	00
*011B	3A	013B	10	015B	00
011C	CC	013C	1C	015C	62
011D	10	013D	C3	015D	49
011E	24	013E	E8	015E	3A
011F	C3	013F	00	015F	

Figure E1-2  
Horizontal axis program (platform)

12. Signify the end of your first program by pressing the **RESET** key.
13. Examine your program for errors using the following procedure:
  - A. Press the **A** key to place the Trainer into the Repeat mode.
  - B. Press the **EXAM** (examine) key. Then enter the program's starting address (0100) to see the first inst/data code.

- C. Compare the displayed inst/data code with that shown for the same address in Figure E1-2.
  - D. If the displayed data is incorrect, press the **CHAN** (change) key and enter the correct inst/data code.
  - E. Press the **FWD** (forward) key. The contents of the next memory address will be displayed. Correct this inst/data code if necessary.
  - F. Continue to step through the program with the **FWD** key, and correct data as necessary. You may step backwards in memory by pressing the **BACK** (backward) key, which decrements memory instead of incrementing it.
  - G. Press the **RESET** key to re-enter the executive mode.
14. Execute the first part of the "Horizontal-Axis" program by first pressing the **A** key, followed by the **DO** key, which results in the following display prompt:
- \_\_\_\_\_dO.
- Then, enter the program's starting address (0100).

NOTE: YOU MAY ABORT ANY PROGRAM BY DEPRESSING EITHER THE **RESET** KEY ON THE MAIN KEYBOARD OR THE **ABORT** KEY NEAR THE EXPERIMENTAL BOARD.

- 15. Observe the Robot's horizontal motion. During this first operation, the Robot will move ahead approximately 6 inches, stop, and then reverse itself to finally stop again near its origin point. The Robot then returns to its executive mode.
- 16. Repeat Step 14, only this time enter starting address 0112.
- 17. Observe the Robot's increased horizontal motion. This time the Robot will travel approximately 6 feet forward, stop, and then reverse itself to once again stop near its origin point.
- 18. Repeat Step 14 using 011C as a starting address.
- 19. Observe the Robot's motion. The platform should move slightly forward and execute a 90 degree right turn.
- 20. Repeat Step 14 using 0123 as a starting address.

21. Observe the Robot's motion. This part of the program causes the platform to reverse back to its beginning point.
22. Repeat Step 14 using 0132 as a starting address.
23. Observe the Robot's motion. The platform should move slightly forward and execute a 90 degree left turn.
24. Repeat Step 14 using 0147 as a starting address.
25. Observe the Robot's motion. This part of the program causes the platform to reverse once again to its beginning point.
26. Use the EXAMine process described in Step 13 to change the inst/data codes from 3A to 01 at memory locations: 0111, 011B, 0122, 0131, 0146, and 0155. These are indicated with an asterisk (\*) in Figure E1-2.
27. Repeat Step 14 using 0100 as a starting address.
28. Observe the Robot's motion. In this final program version, the platform will automatically execute all of its earlier sub-programs.
29. Reposition the Trainer to its original starting point. Initialize its motors as explained in Step 6. Then repeat Steps 27 and 28. Compare the physical positioning to what you observed earlier and note the differences.

## Discussion

Platform mobility is one of the seven axes of motion possible with the ET-100 Trainer. In Experiment 1 your program instructions were limited to simple forward, backward, right, and left motions. Through this experiment, you acquainted yourself with a little of HERO's ultimate capabilities and limitations. As far as base motion is concerned, HERO can move at two additional speeds and even execute 360 degree turns within its own diameter. However, HERO can not easily repeat precision movements with its platform. More complex platform motions are possible, but those will be left to future experiments.

During this first experiment, you learned how to enter, check, modify, and execute instructions through the keyboard. To enter programs you entered the Repeat Mode and specified their position in memory through a starting address. Then you entered the inst/data codes of the program and indicated the program's end by pressing the **RESET** key. Next you checked your entries through HERO's special EXAMine Mode. While in this mode you were able to change any incorrect inst/data codes. The same EXAMine Mode was also used later to modify the entire program's operation.

Entering the Repeat Mode, pressing the A, DO keys and specifying a starting address caused HERO to execute the instructions that you programmed. These basic programming techniques are all fundamental to HERO's operation and will be repeated throughout the remainder of the course.

Another facet of your first experiment involved program subroutines, which we called sub-programs. Although you entered all your inst/data codes during the same programming session, your first program executed initially as separate sub-programs due to strategically placed "stop" codes. These stop codes were originally inserted so you could closely examine HERO's basic platform motions. Later in the experiment, you removed these stop codes causing all the sub-programs to link together. This procedure is one used to test out small portions of a program and will also be used throughout the remaining experiments.

The inst/data codes used in this experiment are only a small subset of HERO's total vocabulary. As you progress through the course, you will learn the actual meaning of all the various types of inst/data codes, as well as how and where to use them. For now, however, you only need to concern yourself with entering the programs we provide and observing the results.

Now, proceed to Experiment 2 where you will familiarize yourself with the remainder of your Robot's axes of motions.

## Experiment 2

### *Robot Familiarization— Manipulator Axes of Motion*



**OBJECTIVES:**

*To identify the seven remaining axes of motion of the ET-18 Robot Trainer.*

*To observe the minimum and maximum amount of travel in each axis of the ET-18's manipulator.*

**Introduction**

In the last experiment, you were introduced to the basics of direct keyboard programming. The program entered familiarized you with HERO's basic platform motions. Through that experiment, you saw one problem inherent to Robot's modest complexity—they have difficulty repeating small precise motions.

The purpose of this experiment is to not only introduce you to the fundamental axes of motions associated with your Robot's head and manipulator, but to also demonstrate the increased accuracy of these motions.

**Material Required**

ET-18 Robot Trainer

**Procedure**

1. Ensure that the Trainer batteries are at full charge.
2. Remove the battery charging and teaching pendant cables from the Trainer.
3. Place the Trainer in the middle of a 6ft by 6ft work area.
4. Enter the inst/data codes of Figure E2-1, following the steps outlined in Experiment 1. Use the starting address shown in the Figure.
5. Press the **RESET** key.

ADDRESS	INST/ DATA	ADDRESS	INST/ DATA	ADDRESS	INST/ DATA
0200	FD	0228	15	*0250	3A
0201	00	0229	D3	0251	D3
0202	00	022A	4C	0252	6C
0203	4D	022B	15	0253	14
0204	00	022C	D3	0254	D3
0205	00	022D	48	0255	68
0206	62	022E	83	0256	2C
0207	49	*022F	3A	0257	D3
0208	D3	0230	D3	0258	6C
0209	C8	0231	28	0259	65
020A	0F	0232	10	025A	D3
020B	D3	0233	D3	025B	68
020C	CC	0234	2C	025C	93
020D	24	0235	10	025D	D3
020E	D3	0236	D3	025E	6C
020F	C8	0237	28	025F	4D
0210	14	0238	65	*0260	3A
0211	D3	*0239	3A	0261	D3
0212	CC	023A	D3	0262	A8
0213	61	023B	28	0263	15
0214	D3	023C	35	0264	D3
0215	C8	*023D	3A	0265	AC
0216	6F	023E	D3	0266	15
0217	FD	023F	88	0267	D3
0218	00	0240	16	0268	A8
0219	00	0241	D3	0269	71
021A	4D	0242	8C	026A	D3
021B	00	0243	16	026B	AC
021C	00	0244	D3	026C	71
021D	62	0245	88	*026D	3A
021E	42	0246	58	026E	FD
*021F	3A	0247	D3	026F	00
0220	D3	0248	8C	0270	00
0221	48	0249	58	0271	4D
0222	05	024A	D3	0272	00
0223	D3	024B	88	0273	00
0224	4C	024C	A5	0274	62
0225	05	024D	D3	0275	49
0226	D3	024E	8C	0276	3A
0227	48	024F	51		

Figure E2-1  
Manipulator axes of motion program.

6. Execute the program from address 0200 using the Repeat Mode and observe the action of the head. Note the different degrees of head motion, ranging from slight movement up to a full swing of 350 degrees.
7. Execute the program from address 0220 using the Repeat Mode and observe the action of the shoulder. The shoulder in your Robot is capable of positioning itself anywhere within a 170 degree arc and can be used for very precise activities. At the end of this sub-program, HERO's arm will be in an upward position.
8. Execute the program from address 0230 using the Repeat Mode and observe the arm as it extends, retracts, and re-extends. This unusual axis of movement helps compensate for HERO's inability to bend towards an object.
9. Execute the program from address 023A using the Repeat Mode and observe the action of both the arm and the wrist. From this part of the experiment, you will note the automatic wrist "yaw" that accompanies a full arm extension. This additional axis of motion heightens the Robots wrist flexibility. Maximum possible wrist "yaw" is limited to about 90 degrees. Wrist yaw is not an independent command, but one associated with arm extension.
10. Execute the program from address 023E using the Repeat Mode and observe HERO's wrist pivot motion. As with the other axes, HERO has a fairly large range of motion. In fact, the wrist can pivot to nearly any position within a 180 degree arc of its starting point. HERO's wrist should extend outward at the end of this sub-program.
11. Execute the program from address 0251 using the Repeat Mode. During this sub-program, HERO's wrist will rotate throughout its range of 355 degrees.
12. Enter the Repeat Mode and execute the sub-program beginning at address 0261. This routine demonstrates the gripper's range of motion. When open, the gripper should be able to reach around and then grasp an object nearly 4 inches in diameter. The final instruction of this sub-program should close the Robot's gripper.
13. Enter the Repeat Mode and execute the final sub-program beginning at address 026E. This final program segment will re-initialize all of the arm and shoulder motors.

14. Link the sub-programs together by changing the inst/data codes from 3A to 01 at these address locations:

021F   022F   0239   023D   0250   0260   026D

15. Using the Repeat Mode, execute your newly linked program starting at address 0200. HERO will execute all of the earlier motions and then automatically re-initialize its arm and shoulder motors. Repeat this procedure and note the increased precision of these motions as compared to those of the main platform motors from Experiment 1.

## Discussion

This experiment showed you the range of motion available to your Robot's head, shoulder, arm, wrist, and gripper. In later experiments, you will not only learn how to program each of these individual motor movements, but you will also see how programs are generated to cause simultaneous operation of various motor combinations.

Now return to your text and proceed with the next Unit.

1. The first step in the process of writing a paper is to choose a topic.

2. The next step is to gather information about the topic.

3. The third step is to organize the information into a logical order.

4. The fourth step is to write the paper.

5. The fifth step is to revise the paper.

6. The sixth step is to proofread the paper.



## Experiment 3

### *On Board Logic Test Probe*

**OBJECTIVES:**

*To familiarize you with breadboarding techniques using the ET-18 on-board experimental board.*

*To demonstrate the operation of a built-in logic probe as a troubleshooting aid.*

**Introduction**

In this experiment, you will construct a logic probe on the ET-18 experimental board. After the logic probe has been constructed, leave it on the experimental board. It will be used in subsequent experiments.

In the first portion of the experiment, you will construct the logic probe. The remaining experiment section demonstrates how the logic probe is used as a valuable troubleshooting aid.

**Material Required****ET-18 Robot Trainer**

- 2    270 ohm, 5 percent, 1/4-watt resistors
- 1    33k ohm, 5 percent, 1/4-watt resistor
- 1    100k ohm, 5 percent, 1/4-watt resistor
- 2    150k ohm, 5 percent, 1/4-watt resistors
- 1    1M ohm, 5 percent, 1/4-watt resistor
- 1    2.2M ohm, 5 percent, 1/4-watt resistor
- 1    2.5V 20mA, red LED
- 1    2.5V, 20mA, green LED
- 1    LM2901 integrated circuit (442-616)

Hookup wire (22 gauge, white, solid conductor)

## Procedure

1. Ensure that the power to the ET-18 Robot Trainer is turned off.
2. Using the parts provided, construct the circuit shown in the schematic diagram of Figure E3-1, on the experimental board. A pictorial diagram of the circuit, showing component placement on the experimental board, is shown in Figure E3-2. NOTE: It is important that the components be placed on the experimental board exactly as shown in Figure E3-2; otherwise, you may not have enough room on the experimental board to connect later experiments.

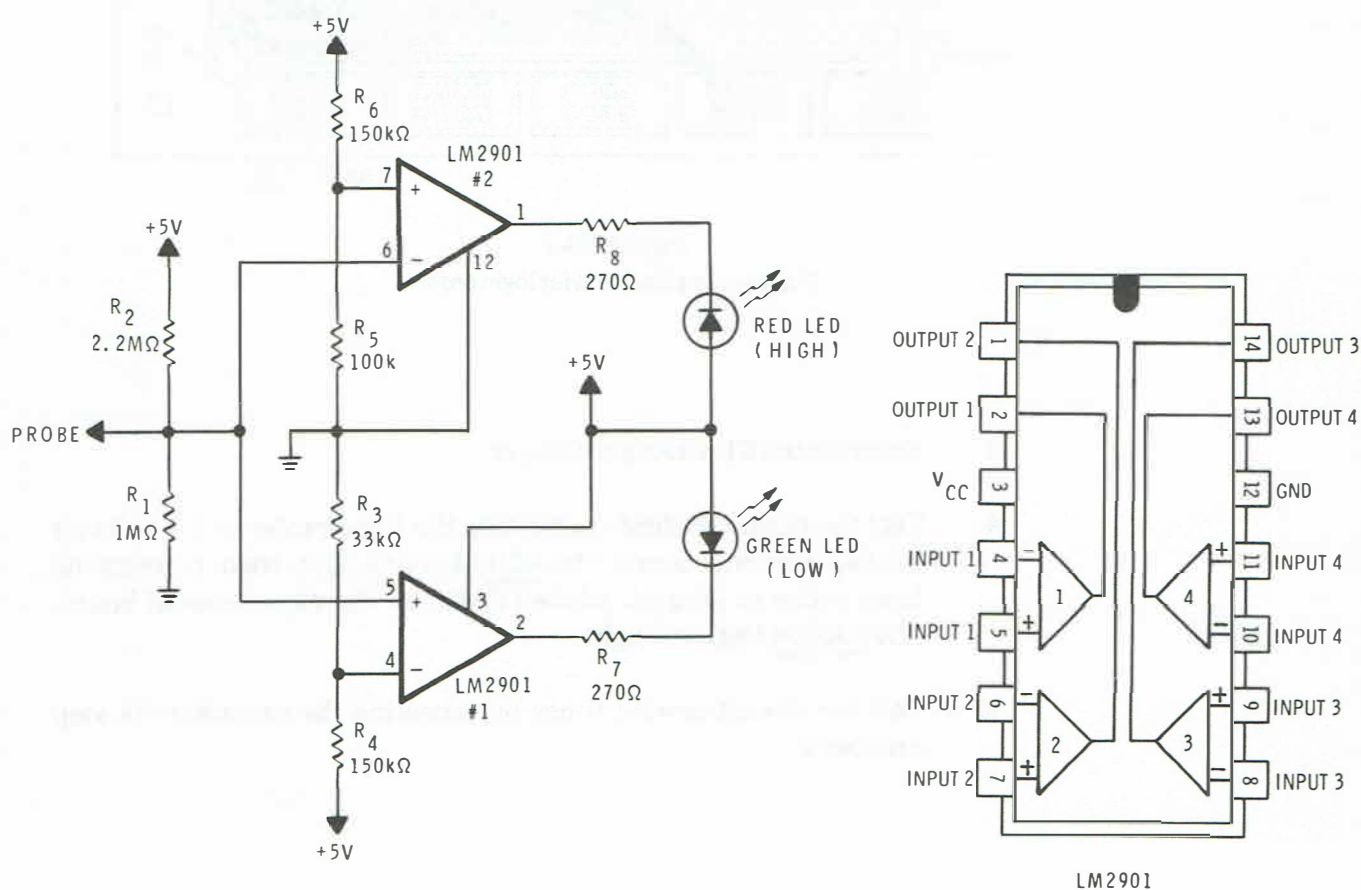


Figure E3-1  
Schematic for logic probe.

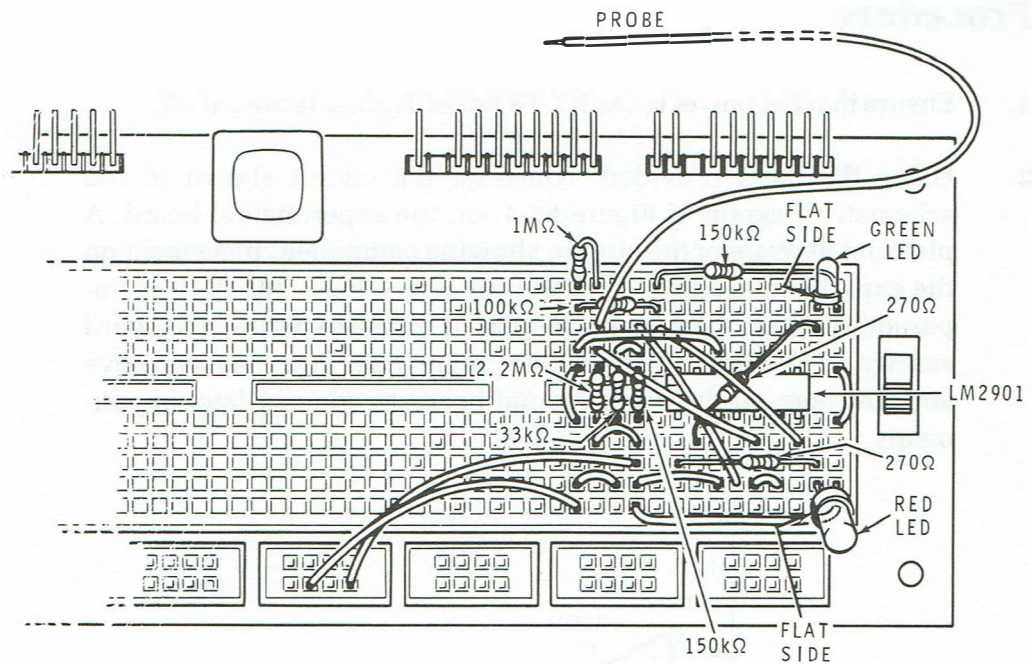


Figure E3-2  
Component placement for logic probe.

3. Energize the ET-18 Robot Trainer.
4. Test the circuit by first connecting the logic probe to +5 volts on the experimental board. The RED LED will light. Next, connect the logic probe to ground, labeled GND, on the experimental board. The GREEN LED will light.
5. Test the circuit several times by repeating the procedure in step number 4.

## Discussion

IC LM2901, shown in Figure E3-1, is a quad single-supply comparator designed for level detection and low-level sensing applications. However, for the logic probe application, only two of the comparators (1 and 2) are actually used.

With the logic probe not being used, a voltage dividing network consisting of R1 and R2 establishes a bias of approximately 1.6 volts on pins 5 and 6 of comparators 1 and 2 respectively. Likewise, a voltage divider consisting of R3 and R4 biases pin 4 of comparator 1 at approximately .9 volts, while voltage divider R5 and R6 biases pin 7 of comparator 2 at approximately 2 volts.

Comparator 1 is connected as a non-inverting device. Thus, in the quiescent state (no input on the logic probe), you have a high output at pin 2 of comparator 1. Recall that when the signal applied to a comparator's non-inverting input (pin 5) is more positive than the signal applied to the inverting input (pin 4), the output is high. With a high output from comparator 1, the green LED can not conduct.

Comparator 2 is connected as an inverting device. Again, with no input from the logic probe, you have a high output at pin 1 of comparator 2. This is because the signal applied to the noninverting input (pin 7) is more positive than the signal applied to the inverting input (pin 6). Thus, with a high output from comparator 2, the red LED will not conduct.

However, when the logic probe is connected to a high logic level (in excess of 2 volts), the inverting input (pin 6) of comparator 2 becomes more positive than noninverting input (pin 7). Because this is an inverting device, a low will be felt at the output. This low at the output will allow the red LED to conduct, showing there is a high logic level present. At the same time, the high logic level is also applied to the noninverting input of comparator 1. This causes pin 5 to become more positive than in the quiescent stage. Thus, the output of comparator 1 will remain high and the green LED will not conduct.



If the logic probe is connected to a low logic level (less than .9 volts), the noninverting input at pin 5 of comparator 1 will become less positive than the inverting input at pin 4. This causes a low to be felt at the output, which in turn causes the green LED to conduct, indicating a low logic level is present. Also, the low logic level on the probe will cause the inverting input at pin 6 of comparator 2 to become less positive than the noninverting input at pin 7. Hence, the output of comparator 2 will be positive and the red LED will not conduct.

As you can see, the on-board logic probe you have constructed will be able to detect low logic levels below + .9 volts and high logic levels above 2 volts. Any measurement between these two values is considered to be invalid. However, these restrictions should not affect the operation of the logic probe as a troubleshooting aid, since the ET-18 uses only low logic levels (0 volts, or ground) or high logic levels (+ 5 volts). The logic probe can also be safely used, however, to measure +12 volts on the ET-18 Robot Trainer.

## Experiment 4

### *Manual Control of a DC Motor*

**OBJECTIVES:**

*To manually control the rotational speed of a DC motor by varying the voltage to the armature circuit.*

*To observe how reversing the polarity of the voltage applied to the armature reverses the directional rotation of the DC motor.*

*To illustrate how motor torque is affected by the amount of voltage applied to the armature.*

*To demonstrate how an increase in motor load affects the amount of current the motor draws.*

**Introduction**

As you learned in Unit 3, there are several methods used to control the speed and directional rotation of a DC motor. In the first portion of this experiment, you will see how the speed of a DC motor is controlled by manually varying the amount of resistance in the armature circuit, thus limiting the amount of voltage applied to the armature. In addition, you will see how the polarity of the signal applied to the armature determines the direction of rotation.

Next, you will investigate how a decrease in the amount of voltage applied to the armature affects the amount of load the motor will carry. Also, you will see how changing the load to a motor causes the armature current to vary.

## Material Required

ET-18 Robot Trainer

Heathkit IM-1210 Digital Multimeter, or equivalent meter

4 15 ohm, 5 percent, 1-watt resistors

1 8-section, SPST, DIP switch

1 36" length, 8-conductor flat cable

1 4-pin connector

2 Large spring connectors

Hockup Wire (22 gauge, White, solid conductor)

## Procedure

1. Make sure that power to the ET-18 Robot Trainer is turned off.
2. Prepare the auxiliary test cable as follows:
  - A. Carefully remove the red and brown conductor pair from the 8-conductor flat cable.
  - B. Observing Figure E4-1A, separate the red and brown conductor pair 1" at one end and 3" at the other end. Remove 3/16" insulation from each conductor.
  - C. Tightly twist together the fine wire strands at each lead end and then melt a small amount of solder on all exposed wire.
  - D. At the 1" end of the conductor pair, cut both conductors so that the exposed portion is 1/8" long.
  - E. Referring to Figure E4-1B, install a spring connector on each of the 1" ends of the conductor pair.
  - F. Insert the spring clips into holes 1 and 4 of the 4-slot connector housing as shown in Figure 12-C.
  - G. Lay the assembled auxiliary test cable aside.



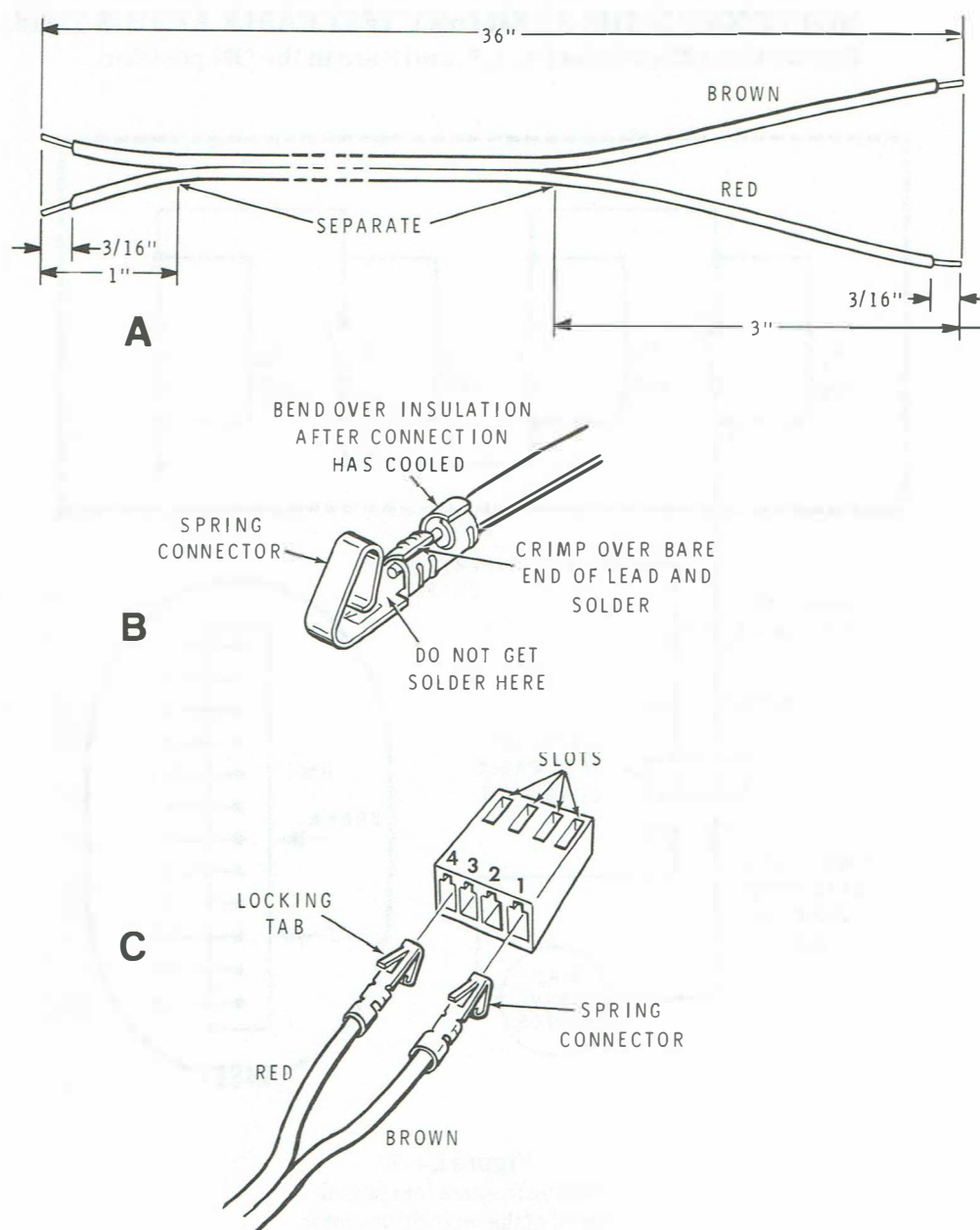


Figure E4-1  
Auxiliary test cable.

3. Refer to Figure E4-2 and, using the four 15 ohm 1-watt resistors and the 8-section DIP switch provided, construct the circuit shown inside the broken lines on the Robot Trainer experimental board. **DO NOT CONNECT THE AUXILIARY TEST CABLE AT THIS TIME.** Ensure that DIP switches 1, 3, 5, and 7 are in the ON position.

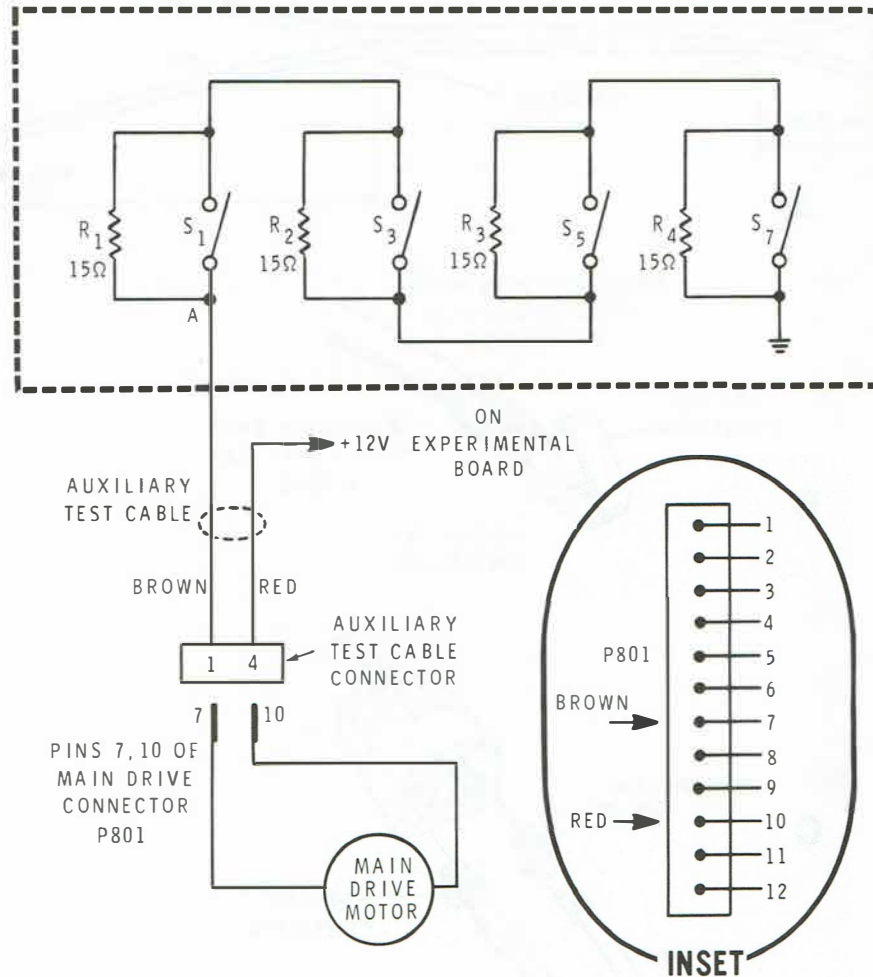


Figure E4-2  
Wiring diagram for manual  
control of the main drive motor.

4. Using wooden blocks, books, etc, elevate the front portion of the Robot Trainer so that the main drive wheel is approximately 1" off the surface of your work area. Check to ensure that the main drive wheel is free to rotate and turn to both its left and right limit switches.

5. Energize and initialize the Robot Trainer.
6. Turn off the power to the Robot Trainer.
7. Observe the location of connector P801 on the Robot Trainer's main drive circuit board. Then carefully remove the main drive circuit board from the Robot Trainer.
8. Refer to the Inset of P801 in Figure E4-2 and connect the auxiliary test cable so that the brown wire mates with pin 7 and the red wire mates with pin 10 of P801 on the Robot Trainer.
9. Connect the other end of the auxiliary test cable so that the brown wire is at point A on the experimental board and the red wire is at +12 volts.
10. Connect the positive lead of your multimeter to point A on the experimental board and the negative lead to ground. Place the meter on the 20 volt range.

NOTE: When you energize the Trainer to perform the remainder of this experiment, the four 15 ohm resistors will generate a significant amount of heat. Therefore, we recommend that you read Steps 11 through 15 prior to proceeding with the experiment. Also, complete these five steps as quickly as possible to prevent damage to the experiment components.

11. Energize the Trainer and observe the direction and speed of wheel rotation. Record the direction of wheel rotation and the voltage shown on the meter.  
Direction of wheel rotation. \_\_\_\_\_  
\_\_\_\_\_volts, all switches on.
12. Place switch 1 in the off position and observe a decrease in the speed of rotation. Record the voltage shown on the meter.  
\_\_\_\_\_volts, switch 1 off.
13. Place switch 3 in the off position and observe another decrease in the speed of rotation. Record the voltage shown on the meter.  
\_\_\_\_\_volts, switch 1 and 3 off.

14. In a similar manner, place switches 5 and 7 in the off position, observing further decreases in the speed of rotation. Record the voltages shown on the meter.

\_\_\_\_\_volts, switches 1, 3, and 5 off.

\_\_\_\_\_volts, all switches off.

15. Turn off power to the Trainer and position all switches to the on position.
16. Switch the connections of the auxiliary test cable at the experimental board so that the red wire is point A and the brown wire is at + 12 volts.
17. Energize the Trainer and again observe the direction of rotation of the main drive wheel. Is the direction of rotation the same as in Step 11?\_\_\_\_\_
18. Turn off power to the Trainer and read the following discussion.

## Discussion

The internal DC resistance of the main drive motor is very low, approximately 2 ohms. Therefore, when the motor is placed in a voltage divider network as shown in Figure E4-3, it has very little effect on the overall resistance of the circuit. As shown in Figure E4-3, when all switches are on (closed), resistors R1, 2, 3, and 4 are bypassed. Thus, the full + 12 volts is applied to the armature of the motor.

However, when switch S1 is opened, R1 is added to the armature circuit of the motor. As you observed in the experiment, the addition of R1 to the circuit caused a voltage drop of approximately 3 volts. Thus, only 9 volts were available to the armature circuit of the motor, decreasing the speed of the motor. In a similar manner, when resistors R2, and R3 were added to the circuit, further decreases in the voltage available to the motor were noted and the speed of motor rotation decreased even more. Finally, when R4 was added to the circuit, the total voltage dropped across the voltage divider network was approximately 10 volts. Hence, very little of the applied voltage was supplied to the motor and the motor barely rotated.

Recall from the text that one method of changing directional rotation of a DC motor is to reverse the polarity of the voltage applied. As shown in the experiment, when you reversed motor leads on the experimental board, the direction of motor rotation also reversed.

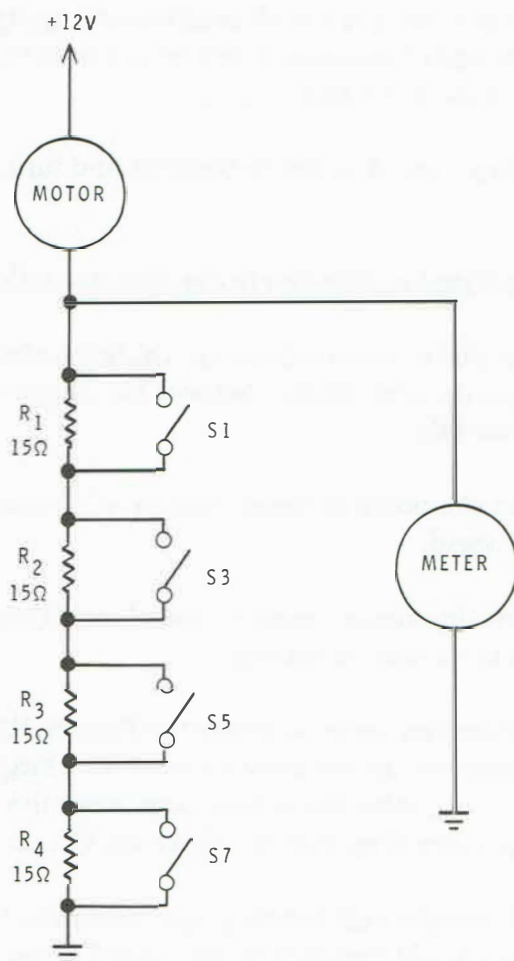


Figure E4-3  
Voltage divider network.



## Procedure (Continued)

19. Energize the Trainer and use your hand to supply a load to the drive motor by applying a slight pressure on the wheel while it is rotating. Does the wheel stop rotating? \_\_\_\_\_
20. Place switches 1 and 3 in the off position and apply approximately the same amount of pressure to the wheel as in the previous step. Does the wheel stop rotating? \_\_\_\_\_
21. Place switches 1 and 3 in the on position and turn off power to the Trainer.
22. Connect the digital multimeter in the circuit as follows:
  - A. Remove the brown auxiliary test cable wire from + 12 volts on the experimental board. Connect the negative meter lead to the brown wire.
  - B. Connect the positive meter lead to + 12 volts on the experimental board.
  - C. Position the meter function switch to DCmA and the range switch to its highest setting.
23. While observing the meter, energize the Trainer. When you first energize the Trainer does the drive motor draw a large amount of current? \_\_\_\_\_. After the initial surge, does the current drop off to an average value of approximately 230mA? \_\_\_\_\_.
24. With the drive motor still running, again supply a load to the motor by applying a slight pressure to the wheel. Does the motor draw more or less current? \_\_\_\_\_.
25. Turn off power to the Trainer, remove the circuit from the experimental board, remove the auxiliary test cable, and replace the main drive circuit board on the Trainer.

## Discussion

As you observed in the last portion of this experiment, the motor provides a greater amount of torque and is thus able to drive a greater load when no external resistance is in the circuit. However, when external resistance is added to the circuit, thus decreasing the voltage available to the motor's armature circuit, not only does the speed of rotation decrease but also the motor's ability to provide torque to the load.

As you learned in the text and observed in the experiment, a DC motor draws more current when it first starts up than it does while it is running. Also, an increase in the load of this DC motor causes the motor to draw more current and, at the same time, slow down. Since this is a permanent magnet DC motor (no field windings), there is no method available by which current to the field windings can be increased, thus increasing the torque capabilities under load. Therefore, a motor of this type would not normally be used when the torque requirements are high or vary from one extreme to the other.

## Discussion

The first of the two main points of the paper is that the current system of international law is not working. The second point is that the current system of international law is not working. The third point is that the current system of international law is not working. The fourth point is that the current system of international law is not working. The fifth point is that the current system of international law is not working. The sixth point is that the current system of international law is not working. The seventh point is that the current system of international law is not working. The eighth point is that the current system of international law is not working. The ninth point is that the current system of international law is not working. The tenth point is that the current system of international law is not working.

The first of the two main points of the paper is that the current system of international law is not working. The second point is that the current system of international law is not working. The third point is that the current system of international law is not working. The fourth point is that the current system of international law is not working. The fifth point is that the current system of international law is not working. The sixth point is that the current system of international law is not working. The seventh point is that the current system of international law is not working. The eighth point is that the current system of international law is not working. The ninth point is that the current system of international law is not working. The tenth point is that the current system of international law is not working.

## Experiment 5

### *Straight Line Programs*

**OBJECTIVES:**

*To identify the purpose of, and differences between, the following opcodes:*

<i>LDA (86<sub>16</sub>)</i>	<i>ADD (8B<sub>16</sub>)</i>	<i>LDA (96<sub>16</sub>)</i>
<i>ADD (9B<sub>16</sub>)</i>	<i>STA (97<sub>16</sub>)</i>	<i>HLT (3E<sub>16</sub>)</i>
<i>CLRA (4F<sub>16</sub>)</i>	<i>INCA (4C<sub>16</sub>)</i>	<i>DECA (4A<sub>16</sub>)</i>

*To enter, examine and single-step through straight-line programs.*

*To examine and then change the contents of the 6808's accumulator and program counter.*

*To add two numbers together using the A accumulator and either immediate or direct instructions.*

*To multiply two numbers through multiple addition using the direct addressing mode.*

*To clear, increment and decrement the A accumulator.*

*To transfer data between two addresses.*

*To locate data storage addresses where they cannot accidentally interfere with proper program execution.*

## **Introduction**

The first half of Unit 4 introduced you to the basic microprocessor and its internal structure. Here, you learned how the microprocessor fetched, interpreted, and executed instructions called opcodes and data called operands. You also learned the differences between immediate, inherent, and direct addressing modes. To help you understand these concepts, you single-stepped through the operation of a simple program.



In this experiment, you will enter, execute, and evaluate the operation of seven different programs. These programs will not only improve your understanding of opcodes, operands, and addressing modes, they will also familiarize you with some techniques used to add, multiply, modify, or store data. Figure E5-1 lists the nine instructions used in this experiment. To help you examine these programs, you will be using the Trainer's special single-step mode. In the single-step mode, you can look at the contents of the accumulator, the program counter, and various memory locations after each instruction is executed. This way, you can follow exactly how the Trainer's computer performs each step.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator (Immediate)	LDA	1000 0110 <sub>2</sub> or 86 <sub>16</sub>	Load the contents of the next memory location into the accumulator.
Add (Immediate)	ADD	1000 1011 <sub>2</sub> or 8B <sub>16</sub>	Add the contents of the next memory location to the present contents of the accumulator. Place the sum in the accumulator.
Load Accumulator (Direct)	LDA	1001 0110 <sub>2</sub> or 96 <sub>16</sub>	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add (Direct)	ADD	1001 1011 <sub>2</sub> or 9B <sub>16</sub>	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator (Direct)	STA	1001 0111 <sub>2</sub> or 97 <sub>16</sub>	Store the contents of the accumulator in the memory location whose address is given by the next byte.
Halt (Inherent)	HLT	0011 1110 <sub>2</sub> or 3E <sub>16</sub>	Stop all operations.
Clear Accumulator (Inherent)	CLRA	0100 1111 <sub>2</sub> or 4F <sub>16</sub>	Reset all bits in the accumulator to 0.
Increment Accumulator (Inherent)	INCA	0100 1100 <sub>2</sub> or 4C <sub>16</sub>	Add 1 to the contents of the accumulator.
Decrement Accumulator (Inherent)	DECA	0100 1010 <sub>2</sub> or 4A <sub>16</sub>	Subtract 1 from the contents of the accumulator.

Figure E5-1

Instructions used in Experiment 5.

Before beginning this experiment, familiarize yourself with the following Trainer keyboard commands:

**DO** - Execute a program, beginning at the address specified after this key is pressed. You can use the **DO** key in either the **DIRECT** Mode or in the **Repeat** Mode. Use the **Direct** Mode for the programs in this experiment.

**EXAM** (examine) - Display the address and memory contents at the address specified after this key is pressed. You can then change the memory contents by pressing the **CHAN**ge key and entering new data. You can use the **EXAM** key in either the **Repeat** or **Direct** Modes.

**FWD** (forward) - Advance to the next memory location and display the contents.

**CHAN** (change) - Open a memory location for write operation. Can only be used in the **EXAM**ine mode.

**BACK** - Go back to the previous memory location and display the contents.

**AUTO** (automatic) - Open the memory location specified, after this key is pressed, so that data can be entered. After data has been entered, automatically advance to the next memory location and wait for data. **AUTO** may be used in either the **Direct** or **Repeat** Modes.

**SS** (single-step) - Go to the address specified by the program counter and execute the instruction at that address. Wait at the next instruction.

**ACCA** (accumulator A) - Display the contents of accumulator A. You may then change the accumulator's contents by pressing the **CHAN** key.

**PC** (program counter) - Display the contents of the program counter. This points to the next location in memory that the microprocessor will "fetch" from. You can change the program counter contents by pressing the **CHAN** key and entering the new address.

**RESET** - Clear any Trainer keyboard commands and enter the **Executive** mode.

## Material Required

ET-18 Robot Trainer

## Procedure

1. The first two programs of this experiment use the immediate addressing mode to add two numbers. Switch your Robot on and press the **RESET** key. You may perform this experiment with the charging unit connected if you wish.
2. Enter the Direct Mode (press the **1** key). Press **AUTO** and enter the program in Figure E5-2, starting at address 0050.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediately with
0051	21	33 <sub>10</sub>	Operand 1.
0052	8B	ADD	Add to accumulator immediately with
0053	17	23 <sub>10</sub>	Operand 2.
0054	3E	HLT	Stop.

Figure E5-2

Addition of two numbers through the immediate addressing mode.

3. Press the **RESET** key. Then examine your program to make sure it was properly entered.
4. Examine the contents of the A accumulator by pressing the **ACCA** key. The initial value should be a random value.
5. Press the **PC** key, then change the contents of the program counter to 0050 (the starting address of your program).
6. Press the **SS** key. This lets the Trainer execute the first instruction. The display should show 00528b. 0052 represents the address of the next instruction; 8b<sub>16</sub> is the next instruction.
7. Press the **ACCA** key and record the value \_\_\_\_\_. The first program instruction was LDA, and the next byte contained the data (operand) to be loaded, which was 21<sub>16</sub>. This should be the value you record in this step.

8. Press the **PC** key and record the value \_\_\_\_\_. This value points to the next memory location, which should be 0052.

You may have noticed that the address 0052 and instruction 8b were displayed when you first pressed the **SS** key. This would seem to indicate that instruction 8b<sub>16</sub> had already been fetched. This is not the case. The 'monitor' program operating from ROM (the same one that allows DO, EXAM, FWD, SS, etc.) allows you to look ahead to the next instruction as part of its program debugging routines.

9. Press the **SS** key and record the value \_\_\_\_\_. The second instruction has now been executed and the display should show the next instruction to be fetched and its address.
10. Press the **ACCA** key and record the value \_\_\_\_\_. The second operand has been added to the first operand and the sum stored in the A accumulator.
11. **DO NOT PRESS THE SS KEY AGAIN.** Although an HLT instruction normally stops program execution, because you are "sharing" the MPU with the Robot, pressing the SS key again may "confuse" the MPU and result in the loss of your program.
12. Enter the program shown in Figure E5-3. Carefully examine the program to make sure it was entered properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediately with operand 1.
0051	40	64 <sub>10</sub>	
0052	8B	ADD	Add to accumulator immediately with operand 2.
0053	0A	10 <sub>10</sub>	
0054	97	STA	Store the sum
0055	58	58 <sub>16</sub>	at this address.
0056	4F	CLRA	Clear accumulator.
0057	3E	HLT	Stop.
0058	00	00	Reserved for data.

Figure E5-3

Addition of two numbers with immediate addressing.



13. Set the program counter to 0050 and single-step through the program. Record the specified information after each step.

Step 1 display \_\_\_\_\_, ACCA \_\_\_\_.

Step 2 display \_\_\_\_\_, ACCA \_\_\_\_.

Step 3 display \_\_\_\_\_, ACCA \_\_\_\_.

Step 4 display \_\_\_\_\_, ACCA \_\_\_\_.

14. Compare your accumulated data with the program in Figure E5-3.

## Discussion

In this segment of the experiment, you added two numbers together using the immediate addressing mode. Remember, in the immediate addressing mode, the operand is part of the instruction. If you wish to change the numbers that you are adding, you must change the instructions within the program. For this reason, the immediate addressing mode is used primarily when dealing with values that will not normally change. These values are called constants. An example of a constant is the freezing point of water, 32°F.

## Procedure (Con't)

15. Enter the program (HEX contents) listed in Figure E5-4.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load accumulator direct with operand 1 which is stored at this address.
0051	57	57 <sub>16</sub>	
0052	9B	ADD	Add to accumulator direct with operand 2 which is stored at this address.
0053	58	58 <sub>16</sub>	
0054	97	STA	Store the sum at this address
0055	59	59 <sub>16</sub>	
0056	3E	HLT	Stop.
0057	20	32 <sub>10</sub>	Operand 1.
0058	17	23 <sub>10</sub>	Operand 2.
0059	00	00	Reserved for sum.

Figure E5-4

Addition of two numbers through the direct addressing mode.



16. Examine the program to make sure it is entered properly. While in the exam mode, press the **ACCA** key and record the value \_\_\_\_\_. This is the value obtained in the previous program, or a value you entered prior to this program.
17. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step. Remember, do not single-step past the HLT instruction.  
  
Step 1 display \_\_\_\_\_. ACCA \_\_\_\_.  
Step 2 display \_\_\_\_\_. ACCA \_\_\_\_.  
Step 3 display \_\_\_\_\_. ACCA \_\_\_\_.
18. Examine address 0059. Its value is \_\_\_\_\_. This value should be identical to the value now stored in the A accumulator.
19. Now compare your recorded data with the program in Figure E5-4. This will give you a general picture of how the microprocessor uses various instructions and data to perform a desired function.
20. Change the data in the A accumulator and at address 0059 to FF. Then execute the program with the DO key beginning with address 0050.
21. The data in the A accumulator is \_\_\_\_\_ and the data in address 0059 is \_\_\_\_\_. These should be the same and equal to the sum of the two operands.
22. The program counter contains the address \_\_\_\_\_. This should be the address of the next memory location after the HLT instruction.

## Discussion

With this program, direct addressing is used to add two numbers together and store the contents in a memory location. If you wish to change the numbers that you are adding, you do not have to alter the program code. (By program code, we mean instructions as opposed to data stored in memory.) Instead, you simply change the contents of the memory locations that contain your numbers.

The direct addressing mode is usually used when the data that you are working with can change from program run to program run. Numbers or values of this type are referred to as variables.

If you wish, you can change the numbers at memory locations 0057 and 0058 and DO the program again. You will see that the program runs perfectly well without changing any of the instructions.

## Procedure (Con't)

23. Load the program shown in Figure E5-5 into the Trainer. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step. This program illustrates how you can multiply a number by another through multiple addition using the direct addressing mode.

Step 1 display \_\_\_\_\_. ACCA \_\_\_\_.

Step 2 display \_\_\_\_\_. ACCA \_\_\_\_.

Step 3 display \_\_\_\_\_. ACCA \_\_\_\_.

Step 4 display \_\_\_\_\_. ACCA \_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load accumulator direct with
0051	59	59 <sub>16</sub>	operand 1 which is stored at this address.
0052	9B	ADD	Add to accumulator direct with
0053	59	59 <sub>16</sub>	operand 1 which is stored at this address.
0054	9B	ADD	Add to accumulator direct with
0055	59	59 <sub>16</sub>	operand 1 which is stored at this address.
0056	9B	ADD	Add to accumulator direct with
0057	59	59 <sub>16</sub>	operand 1 which is stored at this address.
0058	3E	HLT	Stop.
0059	04	04 <sub>10</sub>	Operand 1.

Figure E5-5

Multiplication of a number by another through multiple addition in the direct addressing mode.

24. According to the Trainer's computer, the product of 4 times 4 is \_\_\_\_\_.

25. Enter the program shown in Figure E5-6. This program demonstrates how you can use successive addition in the direct addressing mode to multiply two numbers.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load accumulator direct with
0051	57	57 <sub>16</sub>	operand 1 stored at this address.
0052	9B	ADD	Add to accumulator direct with
0053	57	57 <sub>16</sub>	operand 1 stored at this address.
0054	9B	ADD	Add to accumulator direct with
0055	57	57 <sub>16</sub>	operand 1 stored at this address.
0056	3E	HLT	Stop.
0057	05	05 <sub>10</sub>	Operand 1.

Figure E5-6

Multiplication of two numbers using successive addition in the direct addressing mode.

26. Set the program counter to 0050 and single-step through the program. Record the specified information after each step.

Step 1 display \_\_\_\_\_. ACCA \_\_\_\_.

Step 2 display \_\_\_\_\_. ACCA \_\_\_\_.

Step 3 display \_\_\_\_\_. ACCA \_\_\_\_.

27. Compare your accumulated data with the program in Figure E5-6. As you can see, the program added 05<sub>10</sub> three times ( $5 \times 3$ ), which is one way to multiply the number 5 by 3. More efficient techniques for multiplying will be covered in future experiments.

## Discussion

In these programs, direct addressing is used to multiply a number through repeated addition of that number. Notice that the contents of memory location 0059 do not change no matter how often you access the location. Remember, the contents of a memory location will not change unless you perform an operation that changes them. Simply accessing a location does not affect the location's contents.

## Procedure (Con't)

28. Enter the program listed in Figure E5-7. Use the EXAMine Mode to check your entries.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	4F	CLRA	Clear accumulator.
0051	97	STA	Store the contents
0052	5A	5A <sub>16</sub>	at this address.
0053	4C	INCA	Increment accumulator.
0054	97	STA	Store the contents
0055	5B	5B <sub>16</sub>	at this address.
0056	4A	DECA	Decrement accumulator.
0057	97	STA	Store the contents
0058	5C	5C <sub>16</sub>	at this address.
0059	3E	HLT	Stop.
005A	FF	FF <sub>16</sub>	Reserved for data.
005B	FF	FF <sub>16</sub>	Reserved for data.
005C	FF	FF <sub>16</sub>	Reserved for data.

Figure E5-7

Implementation of the clear, increment, and decrement instructions.

29. Set the program counter to 0050 and single-step through the program. Record the specified information after each step.

Step 1 display \_\_\_\_\_, ACCA \_\_\_\_.  
 Step 2 display \_\_\_\_\_, ACCA \_\_\_\_.  
 Step 3 display \_\_\_\_\_, ACCA \_\_\_\_.  
 Step 4 display \_\_\_\_\_, ACCA \_\_\_\_.  
 Step 5 display \_\_\_\_\_, ACCA \_\_\_\_.  
 Step 6 display \_\_\_\_\_.

30. Compare your accumulated data with the program in Figure E5-7.

## Discussion

This program uses a mix of the inherent, and direct addressing modes. If you look at each of the instructions written in the inherent addressing mode, you'll probably notice that they are one-byte instructions. What may not be so readily apparent is that these instructions do not interface with memory nor do they deal with bytes of data. If you look at these instructions, you will see that they affect only registers within the MPU.

Direct addressing, on the other hand, enables the MPU to "talk" to memory. In order to interface with memory, more information must be available to the MPU. For this reason, instructions written in the direct addressing mode are two bytes in length. The second byte contains the address of the operand.

## Procedure (Con't)

31. Figure E5-8 shows a program to swap the contents of two memory locations. Enter the program and examine the process.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load accumulator direct with operand 1
0051	60	60 <sub>16</sub>	stored at this address.
0052	97	STA	Store operand 1
0053	62	62 <sub>16</sub>	at this address
0054	96	LDA	Load accumulator direct with operand 2
0055	61	61 <sub>16</sub>	at this address.
0056	97	STA	Store operand 2
0057	60	60 <sub>16</sub>	at this address.
0058	96	LDA	Load accumulator direct with operand 1
0059	62	62 <sub>16</sub>	stored at this address.
005A	97	STA	Store operand 1
005B	61	61 <sub>16</sub>	at this address.
005C	4F	CLRA	Clear the accumulator
005D	97	STA	Store the contents
005E	62	62 <sub>16</sub>	at this address.
005F	3E	HLT	Stop.
0060	AA	170 <sub>10</sub>	Operand 1.
0061	BB	187 <sub>10</sub>	Operand 2.
0062	00	00	Temporary storage.

Figure E5-8

Data transfer between two addresses.



32. Set the program counter to starting address 0050 and single-step through the program. Record the specified information after each step.

Step 1 display \_\_\_\_\_, ACCA \_\_\_\_\_.

Step 2 display \_\_\_\_\_, ACCA \_\_\_\_\_.

Step 3 display \_\_\_\_\_, ACCA \_\_\_\_\_.

Step 4 display \_\_\_\_\_, ACCA \_\_\_\_\_.

Step 5 display \_\_\_\_\_, ACCA \_\_\_\_\_.

Step 6 display \_\_\_\_\_, ACCA \_\_\_\_\_.

Step 7 display \_\_\_\_\_, ACCA \_\_\_\_\_.

Step 8 display \_\_\_\_\_, ACCA \_\_\_\_\_.

33. Examine and record the values at addresses:

0010 \_\_\_\_\_.

0011 \_\_\_\_\_.

0012 \_\_\_\_\_.

34. Compare your accumulated data with the program in Figure E5-8.

## Discussion

This program is longer than the previous program but, if you look at it carefully, it is easy to follow the program flow. The operands stored at memory locations 0060 and 0061 are transferred. Memory location 0062 is used as an intermediate storage point for operand 1 while the program moves operand 2 to its new location.

One thing to note is that although the purpose of the program (to transfer two bytes of data) sounds relatively straight-forward, it still requires nine instructions to accomplish. This is because the MPU can do nothing on its own. No matter how simple the task, you must write an instruction for each step. If you remember this fact, it will help you when you attempt to create more complex programs on your own.

The programs entered for this experiment were simple examples of straight-line operations. As such, each instruction, either immediate, direct, or inherent, is executed in sequential order. You executed each program using a special "single-step" Trainer option. By single-stepping your programs, you were able to examine the contents of the accumulator, the program counter, and various memory locations after each instruction was executed.

Single-step program execution is an extremely valuable aid for learning and troubleshooting. In fact, the single-step option is one of the more powerful features of your Robot's computer. As such, it will be used throughout this course.

One important point illustrated by the programs in this experiment involved the placement of data storage. In each program example, data was always stored at addresses beyond that of the HLT instruction. Although this is not an essential programming procedure, it is one very good way to minimize the possibility of accidentally overwriting the initial program. We suggest that you too refrain from placing your data storage addresses within the main program body.

Now proceed to Experiment 6, where you will enter, examine, and single-step your way through programs using various arithmetic and logic instructions.

# Experiment 6

## Arithmetic and Logic Instructions

**OBJECTIVES:**

*To identify the purpose of, and differences between, the following opcodes:*

NEGA	(40 <sub>16</sub> )	SUB	(80 <sub>16</sub> )	SUB	(90 <sub>16</sub> )
ANDA	(84 <sub>16</sub> )	ANDA	(94 <sub>16</sub> )	ORA	(8A <sub>16</sub> )
ORA	(9A <sub>16</sub> )				

*To state how the microprocessor represents signed numbers.*

*To determine the hexadecimal and binary equivalents of negative numbers using two's complement conversion.*

*To subtract numbers in binary.*

*To add numbers of opposite signs in binary.*

*To change binary data from one code to another using OR and AND logic.*

## Introduction

In Experiment 5, you used nine instructions to write various programs. These instructions were:

<u>MNEMONIC</u>	<u>OPCODE</u>	<u>ADDRESSING MODE</u>
LDA	86 <sub>16</sub>	Immediate
LDA	96 <sub>16</sub>	Direct
ADD	8B <sub>16</sub>	Immediate
ADD	9B <sub>16</sub>	Direct
STA	97 <sub>16</sub>	Direct
CLRA	4F <sub>16</sub>	Inherent
INCA	4C <sub>16</sub>	Inherent
DECA	4A <sub>16</sub>	Inherent
HLT	3E <sub>16</sub>	Inherent

Seven new instructions are presented in this experiment. Each is listed in Figure E6-1.

Unit 4 examined the process of binary arithmetic, 2's complement arithmetic, signed number addition, and Boolean logic. Through sample programs, this experiment will illustrate some of the operations presented in Unit 4.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Complement 2's or Negate (Inherent)	NEGA	0100 0000 <sub>2</sub> or 40 <sub>16</sub>	Replace the contents of the accumulator with its complement plus 1.
Subtract (Immediate)	SUB	1000 0000 <sub>2</sub> or 80 <sub>16</sub>	Subtract the contents of the next memory location from the contents of the accumulator. Place the difference in the accumulator.
Subtract (Direct)	SUB	1001 0000 <sub>2</sub> or 90 <sub>16</sub>	Subtract the contents of the memory location whose address is given by the next byte from the present contents of the accumulator. Place the difference in the accumulator.
AND (Immediate)	ANDA	1000 0100 <sub>2</sub> or 84 <sub>16</sub>	Perform the logical AND between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
AND (Direct)	ANDA	1001 0100 <sub>2</sub> or 94 <sub>16</sub>	Perform the logical AND between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.
OR, Inclusive (Immediate)	ORA	1000 1010 <sub>2</sub> or 8A <sub>16</sub>	Perform the logical OR between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
OR, Inclusive (Direct)	ORA	1001 1010 <sub>2</sub> or 9A <sub>16</sub>	Perform the logical OR between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.

Figure E6-1

Instructions introduced in this experiment.



## Materials Required

ET-18 Robot Trainer

## Procedure

1. In the first part of the experiment, you will see how the microprocessor represents negative and positive numbers. The program shown in Figure E6-2 loads a positive number into the accumulator and then repeatedly decrements the number until it is negative. Enter this program into the Trainer. Verify that you entered it properly by examining each address.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediate with 05.
0051	05	05	
0052	4A	DECA	Repeatedly decrement the accumulator.
0053	4A	DECA	
0054	4A	DECA	
0055	4A	DECA	
0056	4A	DECA	
0057	4A	DECA	
0058	4A	DECA	
0059	4A	DECA	
005A	4A	DECA	
005B	4A	DECA	
005C	4A	DECA	Halt
005D	4A	DECA	
005E	3E	HLT	

Figure E6-2

This program decrements the contents of the accumulator from +5 to -8.

2. Go to the single-step mode by: pressing the **PC** key; pressing the **CHAN** key; and entering the starting address (0050). Single-step through the program by repeatedly pressing the **SS** key. Notice that the first instruction places  $+5_{10}$  in the accumulator. Refer to Figure E6-3 and record the contents of the accumulator after each DECA instruction is executed. Convert the accumulator contents to binary and record them in Figure E6-3.
3. In Step 7, the number in the accumulator changed from 0 to -1. The microprocessor expresses  $-1$  as  $-16$  or  $-2$ . The table you have developed in Figure E6-3 shows how the microprocessor expresses the signed number from  $+5$  to  $-5$  in both hexadecimal and binary. The next program will add signed numbers like these.

AFTER STEP	CONTENTS OF ACCUMULATOR		
	DECIMAL	HEXADECIMAL	BINARY
1	+5	05	0000 0101
2	+4		
3	+3		
4	+2		
5	+1		
6	0		
7	-1		
8	-2		
9	-3		
10	-4		
11	-5	FB	1111 1011

Figure E6-3

Record results here.

4. Enter the program shown in Figure E6-4. Use the single step mode to execute the program. What number is in the accumulator after the first instruction is executed?  $_{-16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediate
0051	05	+5	with +5.
0052	8B	ADD	Add immediate
0053	FB	-5	-5.
0054	8B	ADD	Add immediate
0055	FC	-4	-4
0056	3E	HLT	

Figure E6-4  
Adding signed numbers.

5. What number is in the accumulator after the second instruction is executed?  $_{-16}$  or  $_{-2}$ . What decimal number does this represent? \_\_\_\_\_.
6. What number is in the accumulator after the third instruction is executed?  $_{-16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.

## Discussion

These examples illustrate how the microprocessor represents signed numbers. Further experiments will show that the microprocessor can represent signed numbers between  $+127_{10}$  and  $-128_{10}$ . You can determine the bit pattern for each negative number by clearing the accumulator and decrementing the required number of times. However, there are much simpler ways of determining the proper bit pattern for negative numbers.

The simplest way is to start with the positive binary equivalent and create the two's complement value by changing all 0's to 1's and 1's to 0's and adding 1. The microprocessor has an instruction that will do this for us. It is called the two's complement or Negate instruction. Its mnemonic is NEGA. This instruction changes the number in the accumulator to its two's complement. It is used to change the sign of a number. An example of this instruction's use is given in the next portion of this experiment.

## Procedure (Continued)

7. Load the program shown in Figure E6-5. Use the single-step mode to execute the program. Execute the first instruction by depressing the SS key. What number is in the accumulator?  $-_{16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.
8. Execute the second instruction. What number is in the accumulator?  $-_{16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_. Compare this with the number in step 7. What affect did the NEGA instruction have? \_\_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediate
0051	05	+ 5	with + 5.
0052	40	NEGA	Change the number to -5.
0053	40	NEGA	Change it back to + 5.
0054	4A	DECA	Decrement the number to + 4.
0055	40	NEGA	Change the number to -4.
0056	40	NEGA	Change it back to + 4.
0057	3E	HLT	Halt

Figure E6-5

Using the NEGA instruction.

9. Execute the third instruction. What number is in the accumulator?  $-_{16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_. Is your answer the same as that found in Step 7? \_\_\_\_\_.
10. Execute the fourth instruction. This decrements the accumulator so that it now contains the signed decimal number \_\_\_\_\_.
11. Execute the fifth instruction. What number is in the accumulator?  $-_{16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.
12. Execute the sixth instruction. The number in the accumulator is  $-_{16}$  once more.

## Discussion

The program used the NEGA instruction four times. The first time, the NEGA instruction changed  $05_{16}$  to its two's complement  $FB_{16}$ . Referring back to the table you developed in Figure E6-3, this is the representation for  $-5_{10}$ . Thus, the NEGA instruction effectively changes the sign of the number in the accumulator. The next step proved this again by converting  $-5_{10}$  back to  $+5_{10}$ . To further emphasize the point, the number was decremented to  $+4_{10}$ . The next NEGA instruction changed this to  $FC_{16}$  which is the representation for  $-4_{10}$ . The final NEGA instruction converts this back to  $+4_{10}$ . This instruction allows us to convert a positive number to its negative equivalent and vice versa.

In Unit 4, you learned that the MPU can work with signed numbers in the range of  $+127_{10}$  to  $-128_{10}$  or unsigned numbers in the range of 0 to  $255_{10}$ . This capability results from the way we interpret bit patterns. The following steps will demonstrate this.

## Procedure (Continued)

13. Figure E6-6 shows a program for adding the unsigned numbers  $220_{10}$  and  $27_{10}$ . Load this program into the Trainer and execute it. The final result in the accumulator is  $_{-16}$  or  $_{-2}$ . What unsigned decimal number does this represent? \_\_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediate
0051	DC	$220_{10}$	with $220_{10}$ .
0052	8B	ADD	Add immediate
0053	1B	$27_{10}$	$27_{10}$ .
0054	3E	HLT	Halt.

Figure E6-6

Adding unsigned numbers.



14. Figure E6-7 shows a program for adding the signed numbers  $-36_{10}$  and  $27_{10}$ . Load and execute this program. The final result in the accumulator is  $-16$  or  $-----2$ . What signed decimal number does this represent? \_\_\_\_\_.
15. Compare the results obtained in Steps 13 and 14. Compare the HEX Contents columns of Figure E6-6 with that of Figure E6-7.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediate
0051	DC	$-36_{10}$	with $-36_{10}$ .
0052	8B	ADD	Add immediate
0053	1B	$+27_{10}$	$+27_{10}$ .
0054	3E	HLT	Halt.

Figure E6-7  
Adding signed numbers.

## Discussion

This demonstrates that the MPU simply adds bit patterns. It is our interpretation of these patterns that decides whether we are using signed or unsigned numbers. After all, the two programs are identical except for our interpretation of the input and output data.

Negative numbers are often encountered when performing subtract operations. The SUB instruction, written in the immediate mode, is demonstrated in the next program.

## Procedure (Continued)

16. Load the program shown in Figure E6-8. Execute the program using the single-step mode. What is the number in the accumulator after the first subtract instruction is executed?  $-16$  or  $-----12$  or  $-10$ .
17. What is the number in the accumulator after the second subtract instruction is executed?  $-16$  or  $-----2$ . What signed decimal number does this represent? \_\_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load accumulator immediate
0051	2F	$47_{10}$	with $47_{10}$ .
0052	80	SUB	Subtract immediate
0053	10	$16_{10}$	$16_{10}$ .
0054	80	SUB	Subtract immediate
0055	23	$35_{10}$	$35_{10}$ .
0056	3E	HLT	Halt.

Figure E6-8

Using the subtract instruction.

## Discussion

The first subtract instruction subtracted  $16_{10}$  from  $47_{10}$ , leaving  $31_{10}$ . The second one subtracted  $35_{10}$  from  $31_{10}$ . This produced a result of  $-4_{10}$ . However, the MPU expressed  $-4$  in two's complement form ( $FC_{16}$  or  $1111\ 1100_2$ ). You will find this to be the case anytime the MPU produces a negative result.

Now let's look at some of the logical instructions available to the microprocessor. The AND and OR instructions are described in Figure E6-1. Carefully read the description of these instructions given there. While these instructions have many uses, we will demonstrate only one here.

Earlier you learned that certain peripheral devices communicate with computers using the ASCII code. Thus, when the "2" key on a teletypewriter is pushed, the computer receives the ASCII code for 2, which is  $0011\ 0010$ . The ASCII code for 6 is  $0011\ 0110$ . Notice that the four least significant bits of the ASCII character are the binary value of the corresponding numeral. Thus, we can convert the ASCII characters for

the numerals 0 through 9 to binary simply by setting the four most significant bits to 0's. Likewise, we can convert the binary numbers 0000 0000 through 0000 1001 to ASCII by changing the four most significant bits to 0011.

In the following program, we will use the ANDA and ORA instructions to perform these conversions.

### Procedure (Continued)

18. Load the program shown in Figure E6-9. Single-step through the first instruction. The number in the accumulator is \_\_\_\_\_<sub>2</sub>.
19. Execute the second instruction. This AND's the contents of the accumulator with the value 0F<sub>16</sub>. The number in the accumulator after this ANDA operation is \_\_\_\_\_<sub>2</sub>. Compare this with the number that was in the accumulator in Step 18. Now compare both numbers with 0F<sub>16</sub>. A 1 in the original number is retained only if there is a \_\_\_\_\_ in the corresponding bit position of the mask.
20. Execute the third instruction. In what memory location is the number in the accumulator stored? \_\_\_\_\_<sub>16</sub>. What number is now in the accumulator? \_\_\_\_\_<sub>2</sub>. Does the number still appear in the accumulator after being stored in memory?  
\_\_\_\_\_

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load the accumulator with
0051	5B	5B	the ASCII character at this address.
0052	84	ANDA	AND it with
0053	0F	0F	this value.
0054	97	STA	Store the binary equivalent
0055	5C	5C	at this address.
0056	8A	ORA	OR the number with
0057	30	30	this value.
0058	97	STA	Store the result
0059	5D	5D	here.
005A	3E	HLT	Stop
005B	37	0011 0111	ASCII character for numeral 7.
005C	—	—	Reserved
005D	—	—	Reserved

Figure E6-9  
Using the AND and OR instruction.

21. Execute the fourth instruction. This OR's the contents of the accumulator with the  $30_{16}$  - - - - -  $_2$ . The number in the accumulator is - - - - -  $_2$ . Compare this with  $30_{16}$  and the number that was in the accumulator in Step 20. A 1 is produced in the result whenever there is  $\odot$  - - - - - in the corresponding bit position of either the original number, the mask, or both.
22. Execute the fifth instruction. This stores the number in memory location - - - - -  $_{16}$ .
23. Examine memory locations  $005B_{16}$ ,  $005C_{16}$ , and  $005D_{16}$  and compare their contents.

## Discussion

The program first converts the ASCII code for the number "7" to the binary number 0000 0111. It does this by ANDing the ASCII code with the number 0000 1111<sub>2</sub>. Notice that a 1 bit in  $0F_{16}$  allows the corresponding bit in the original number to be retained. The four most significant bits of the original number are changed to zero because they are ANDed with 0's.

The OR operation restores the ASCII character by attaching 0011 as the four most significant bits.

As used here, the process of altering a number by performing a logical AND or OR operation is called "masking." The "mask" is the operand of the AND or OR instruction. The "mask," along with a logical instruction, is used to change the contents of the accumulator.

# Experiment 7

## *Program Branches*



**OBJECTIVES:**

*To manipulate the N, Z, V, and C condition code registers and determine the conditions that set and reset these flags.*

*To verify the operation of a simple multiply by repeated addition program that uses the BEQ conditional branch instruction and the BRA instruction.*

*To demonstrate the ability to write a program that divides by repeated subtraction and uses a conditional branch and BRA instruction.*

*To introduce a shorthand method of calculating relative addresses.*

*To verify the operation of a program that converts BCD numbers to their binary equivalent.*

*To demonstrate the effect an incorrect relative address can have on a program operation and how the microprocessor can be used to debug programs.*

## Introduction

As we mentioned previously, conditional branch instructions give the computer the power to make decisions. As the name implies, a certain condition must be met before a branch takes place. The condition code registers monitor the accumulator and signal the presence of a specific condition. If the MPU encounters a conditional branch instruction, it checks the condition code registers, or flags, to see if the condition is satisfied. If the specific condition is satisfied, the program branches off to another section. If not, the normal program continues.

Therefore, the decisions of the conditional branch instructions are based on the contents of the condition code registers. A sound knowledge of how these flags are set and cleared will enhance your ability as a programmer.

Since condition code registers are very important, your Trainer was designed with a special key to allow you to examine these flags. The key is labelled "CC" for "Condition Code." When this key is pressed, the state of the condition code registers will be displayed. Each LED displays the contents of one register. Figure E7-1 denotes the corresponding register on the Trainer. You may want to mark these registers on your Trainer, using a small piece of tape.

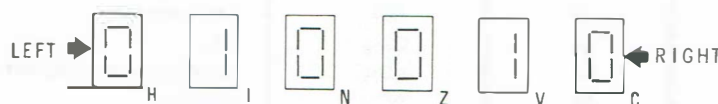


Figure E7-1  
Displaying the conditions  
of the flags.

Notice that there are six flag registers. For the moment we aren't concerned with the two left-most flags. They will be covered in a later unit. However, we are interested in the N, Z, V, and C flags, because they indicate conditions that can lead to conditional branches. Notice that the flags can either be set as indicated by a 1 or they can be cleared as indicated by a 0.

In this first portion of the experiment, you will implement a "do-nothing" program that manipulates the condition code registers. Then single-stepping through the program, you will examine how the contents of the accumulator cause these flags to change.

## Material Required

ET-18 Robot Trainer

## Procedure

1. Turn on the Trainer and then press the **RESET** key.
2. Load the program listed in Figure E7-2 into the Trainer. Once the program is loaded, go back and examine it to insure that it's entered correctly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	01	NOP	"DO Nothing" Instruction
0051	86	LDA	Load the accumulator immediate
0052	FF	FF <sub>16</sub>	with FF <sub>16</sub> .
0053	86	LDA	Load the accumulator immediate
0054	77	77 <sub>16</sub>	with 77 <sub>16</sub> .
0055	86	LDA	Load the accumulator immediate
0056	00	00 <sub>16</sub>	with 00 <sub>16</sub> .
0057	86	LDA	Load the accumulator immediate
0058	01	01 <sub>16</sub>	with 01 <sub>16</sub> .
0059	86	LDA	Load the accumulator immediate
005A	92	92 <sub>16</sub>	with 92 <sub>16</sub> .
005B	8B	ADD	Add immediate
005C	C6	C6 <sub>16</sub>	C6 <sub>16</sub> .
005D	86	LDA	Load the accumulator immediate
005E	08	08 <sub>16</sub>	with 08 <sub>16</sub> .
005F	8B	ADD	Add immediate
0060	08	08 <sub>16</sub>	08 <sub>16</sub> .
0061	86	LDA	Load the accumulator immediate
0062	01	01 <sub>16</sub>	with 01 <sub>16</sub> .
0063	80	SUB	Subtract immediate
0064	02	02 <sub>16</sub>	02 <sub>16</sub> .
0065	86	LDA	Load the accumulator immediate
0066	77	77 <sub>16</sub>	with 77 <sub>16</sub> .
0067	80	SUB	Subtract immediate
0068	66	66 <sub>16</sub>	66 <sub>16</sub> .
0069	86	LDA	Load the accumulator immediate
006A	49	49 <sub>16</sub>	with 49 <sub>16</sub> .
006B	8B	ADD	Add immediate
006C	60	60 <sub>16</sub>	60 <sub>16</sub> .
006D	86	LDA	Load the accumulator immediate
006E	10	10 <sub>16</sub>	with 10 <sub>16</sub> .
006F	3E	HLT	Halt.

Figure E7-2

Program to illustrate the condition code registers.

Now look at the first instruction of the program in Figure E7-2. It has the opcode 01 and the mnemonic is "NOP". As the comments column points out, this is a "do-nothing" type of instruction called a "No-Op." In other words, it performs no operation. In this program, the NOP's primary function is to allow you to see the first instruction before it's executed.

In previous experiments, you probably noticed that when you single-stepped through programs, you never saw the first instruction. This is because in the "SS" mode, the Trainer executes the first instruction automatically and then stops on the second instruction. This can be somewhat confusing.

To offset this problem, we merely insert the NOP. The Trainer "sees" this as the first instruction, although nothing is accomplished by the NOP. Therefore, the Trainer displays the next instruction, which is the first "real" instruction of the program, permitting you to view it before it's executed.

3. Load the program counter with address 0050 and then press the **SS** key. Recall that the first four displays represent the address that's currently in the program counter. The two right-most displays show the opcode stored at this address. Record the information below.

PC\_\_\_\_OP CODE\_\_

Now, press the **ACCA** key and record the contents of the accumulator.

ACCA\_\_

The contents of the accumulator will be a random number, since we haven't yet executed a program instruction.

Now, press the **CC** key and record the contents of the N, Z, V, and C condition code registers below.

----

NZVC

Again, the states of the flags are random at this time.

4. Now, press the **SS** key and then the **ACCA** key. Record the contents of the accumulator below.

ACCA\_\_

Press key **CC** and record the state of the N flag below.

          
N Z V C

With the negative number  $FF_{16}$  in the accumulator, the negative (N) flag is set. Incidentally, any number that has a 1 in bit 7 may be considered a negative number.

5. Press the **SS** key again. The program count should now be  $0055_{16}$  and the opcode at this address is 86. Now check and record the contents of the accumulator and the N flag.

ACCA\_\_              
          N Z V C

With the positive number  $77_{16}$  in the accumulator, the N flag is cleared, or reset, to 0.

From the information gathered in Steps 4 and 5, what conclusions do you reach with respect to the N flag and the contents of the accumulator?

6. Single-step the program again. The program count is now  $0057_{16}$ . Record the contents of the accumulator and the condition of the Z flag below.

ACCA\_\_              
          N Z V C

With  $00_{16}$  in the accumulator, the Z flag is set.

Press **SS** and again record the contents of the accumulator and the Z flag below.

ACCA\_\_              
          N Z V C



The accumulator now contains  $01_{16}$  and the Z flag is cleared. What is the relation between the contents of the accumulator and the Z, or zero flag?

---

7. Single-step again and record the information below.

ACCA \_\_       
NZVC

This step loads the number  $92_{16}$  into the accumulator. Bit 7 of the accumulator contains a  $1_2$  so the N flag is set. Naturally, the Z flag is cleared. The next instruction will add  $C6_{16}$  to the contents of the accumulator. As shown below, this operation should generate a carry.

1001	0010	=	$92_{16}$
1100	0110	=	$C6_{16}$
0101	1000	=	$158_{16}$

CARRY      <sup>1</sup>  
↑

Press the **SS** key and record the information below.

ACCA \_\_       
NZVC

The 8-bit accumulator cannot hold the 9-bit sum. However, the carry generated by the addition sets the C flag.

8. This step loads the number  $08_{16}$  into the accumulator. Press the **SS** key and record the information below.

ACCA \_\_       
NZVC

Notice that loading this new number into the accumulator didn't affect the carry (C) flag. The next step will add  $08_{16}$  to the contents of the accumulator ( $08_{16}$ ).

9. Press the **SS** key and record the information below.

ACCA \_\_       
NZVC

The accumulator now contains the sum of the addition ( $10_{16}$ ) and the carry flag is cleared.

From the results of Steps 8 and 9, you might conclude that the carry flag can be cleared by another \_\_\_\_\_ that does not result in a carry.

10. Press the **SS** key. The program count should now be  $0063_{16}$ . Record the information below.

ACCA \_\_       
NZVC

This shows that the accumulator contains  $01_{16}$  and that the N, Z, and C flags are all cleared. When the next instruction is executed, the number  $02_{16}$  will be subtracted from  $01_{16}$  (the contents of the accumulator). As shown below, the subtraction should result in a borrow, setting the C flag.

Borrow	↑				
		0000	0001	=	$01_{16}$
		0000	0010	=	$02_{16}$
		1111	1111	=	$FF_{16}$

Notice that the difference is  $FF_{16}$ . This will \_\_\_\_\_ the N flag.  
set/clear

11. Press the **SS** key and record the information below.

ACCA \_\_       
NZVC

As expected, the difference produced is  $FF_{16}$ . Also, the N flag is set, indicating a negative number is in the accumulator and the C flag indicates a borrow occurred.

The next step will execute the instruction that loads  $77_{16}$  into the accumulator. After this LDA operation, the C flag will be \_\_\_\_\_.  
set/cleared

12. Press the **SS** key and record the information below.

ACCA\_\_    \_\_\_\_  
          NZVC

Notice that the C flag is still set and that  $77_{16}$  is in the accumulator. Now we will subtract  $66_{16}$  from the accumulator contents ( $77_{16}$ ).

Press the **SS** key and record the information below.

ACCA\_\_    \_\_\_\_  
          NZVC

The difference ( $11_{16}$ ) is now stored in the accumulator and, since no borrow is generated, the C flag is cleared.

13. In this step, the first instruction loads the accumulator with the number  $49_{16}$ . The next instruction adds the number  $60_{16}$  to  $49_{16}$ . As shown below, the addition of these numbers causes an overflow into the sign bit (bit 7) and the sum,  $A9_{16}$ , appears to be a negative number.

0100	1001	=	$49_{16}$
0110	0000	=	$60_{16}$
<hr/>			
1010	1001	=	$A9_{16}$

Overflow changes  
sign bit.

Of course, this is incorrect and the MPU must be notified of this overflow. This is the purpose of the V flag.

Press the **SS** key and record the information below.

ACCA\_\_    \_\_\_\_  
          NZVC

The number  $49_{16}$  is in the accumulator and the N, Z, V, and C flags are cleared.

Single-step once more and then record the information below.

ACCA \_\_    \_\_\_\_  
          NZVC

The sum  $A9_{16}$  is now in the accumulator. Notice that the N and V flags are set, indicating that the number in the accumulator is negative and that an overflow occurred.

14. When the next instruction is executed, the number  $10_{16}$  will be loaded into the accumulator.

Single-step the program and record the information below. Notice that the opcode 3E (a halt) is the next instruction, so the program is finished.

ACCA \_\_    \_\_\_\_  
          NZVC

The accumulator contains the number  $10_{16}$ , and all flags cleared.

From this, you might conclude that any instruction that doesn't produce an overflow in the accumulator will \_\_\_\_\_ the V flag.  
set/clear

## Discussion

In this portion of the experiment, you stepped through a simple program that manipulated the condition code registers. In Step 4, the negative number  $FF_{16}$  was loaded into the accumulator. This set the N flag to 1<sub>2</sub>. In Step 5, the positive number  $77_{16}$  was loaded into the accumulator. And, as you noted, the N flag was cleared or reset to 0<sub>2</sub>. From these two steps you should have concluded that when the number in the accumulator is negative, the N flag is set. And when the accumulator contains a positive number, the N flag is cleared.

In Step 6, the accumulator was loaded with  $00_{16}$ . This set the Z flag to 1<sub>2</sub>. Next, when  $01_{16}$  was loaded, the Z flag was reset or cleared to 0<sub>2</sub>. Your conclusion should have been that when the accumulator contains  $00_{16}$ , the Z flag is set. If it contains any number other than  $00_{16}$ , the Z flag is cleared.

Next, you examined the C flag. When a carry was generated by the addition of the two numbers,  $92_{16}$  and  $C6_{16}$ , the C flag was set. In Step 8, you noted that merely loading a new number into the accumulator did not clear the C flag. The carry flag was cleared by another addition that did not result in a carry. Your conclusion should have been that the C flag can only be cleared by an arithmetic operation that does not result in a carry.

As you proved in Steps 10 and 11, a subtraction that results in a borrow also sets the C flag. Again, the C flag was cleared by an arithmetic operation, in this case a subtraction, that did not generate a borrow. Therefore, the C flag can only be cleared or reset to 0<sub>2</sub> by an arithmetic operation that does not result in a borrow or carry.

You concluded this phase of the experiment by adding two positive numbers, the sum of which overflowed into the sign bit of the accumulator indicating a negative number. This set the V or overflow flag, showing that the sum should not be a negative number as the N flag indicated. The next LDA instruction cleared the V flag. From this, you should conclude that the V flag is cleared by any instruction that doesn't produce an overflow.

In the next sections of this experiment, you will step through a few branching programs that illustrate the use of the branch always (BRA) instruction and certain conditional branch instructions. These branch instructions were discussed in Unit 5. We'll begin with the multiply by repeated addition program.



## Procedure (Continued)

15. Enter the program listed in Figure E7-3 into the Trainer. This program multiplies  $05_{16}$  and  $02_{16}$  and stores the product in address  $0063_{16}$ . Recheck the program to insure that it's entered correctly.
16. Notice that the program contains two branch instructions: the BEQ (Branch if Equal Zero) at address  $0055_{16}$  and the BRA (Branch Always) at address  $005E_{16}$ .

The branch if equal zero (BEQ) instruction implies by its name that a conditional branch will occur when the \_\_\_\_\_ flag is set.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	4F	CLRA	Clear the accumulator.
0051	97	→ STA	Store the product
0052	63	63	in location $63_{16}$ .
0053	96	LDA	Load the accumulator with the
0054	62	62	multiplier from location $62_{16}$ .
0055	27	BEQ	If the multiplier is equal to zero,
0056	09	09	branch down to the Halt instruction.
0057	4A	DECA	Otherwise, decrement the multiplier.
0058	97	STA	Store the new value of the
0059	62	62	multiplier back in location $62_{16}$ .
005A	96	LDA	Load the accumulator with the
005B	63	63	product from location $63_{16}$ .
005C	9B	ADD	Add
005D	61	61	the multiplicand to the product.
005E	20	BRA	Branch back to instruction
005F	F1	F1	in location 51.
0060	3E	→ HLT	Halt.
0061	05	05	Multiplicand.
0062	02	02	Multiplier.
0063	—	—	Product.

Figure E7-3

Program to multiply by repeated addition.

17. Now, set the program counter to 0050 and single-step through the program, recording the information in the chart of Figure E7-4. Notice that you will be monitoring the Z flag. A comments column is provided so you can make notes about each step. Use the program listing as a reference for each opcode and the corresponding operand.
18. When the BEQ instruction is executed and the Z flag is set, the program branches to the \_\_\_\_\_ instruction.

When the multiplier was  $02_{16}$ , the program halted on the \_\_\_\_\_ pass through the program.

If the multiplier is changed to  $06_{16}$ , how many passes would the program make before it halts? \_\_\_\_\_.

19. Examine the contents of address  $0063_{16}$  and record below.

0063 \_\_\_\_.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure E7-4  
Single-stepping through the Multiply by repeated addition program.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0051	97	00	1	Store the product (00 <sub>16</sub> ) in address 0063 <sub>16</sub> .
2	0053	96	00	1	Load the accumulator with the multiplier (02 <sub>16</sub> ) from address 0016 <sub>16</sub> .
3	0055	27	02 ↑ Multiplier	0	BEQ. Check the Z flag. It's not set so continue.
4	0057	4A	02	0	Decrement the multiplier (02 <sub>16</sub> ).
5	0058	97	01 ↑ New Multiplier	0	Store the new multiplier (01 <sub>16</sub> ) at address 0062 <sub>16</sub> .
6	005A	96	01	0	Load the accumulator with the product (00) at address 0063 <sub>16</sub> .
7	005C	9B	00	1	Add the multiplicand (05) giving new product.
8	005E	20	05 ↑ New Product	0	Branch back to address 0051 <sub>16</sub> .
9	0051	97	05	0	Store the product (05 <sub>16</sub> ) in address 0063 <sub>16</sub> .
10	0053	96	05	0	Load the accumulator with the multiplier (01 <sub>16</sub> ) located at address 0062 <sub>16</sub> .
11	0055	27	01	0	BEQ. Check Z flag. It's not set so continue.
12	0057	4A	01	0	Decrement the multiplier (01 <sub>16</sub> ).
13	0058	97	00 ↑ New Multiplier	1	Store the new Multiplier (00 <sub>16</sub> ) at address 0062 <sub>16</sub> .
14	005A	96	00	1	Load the accumulator with the product (05 <sub>16</sub> ) at address 0063 <sub>16</sub> .
15	005C	9B	05	0	Add the multiplicand (05 <sub>16</sub> ) giving new product.
16	005E	20	0A ↑ New Product	0	Branch back to address 0051 <sub>16</sub> .
17	0051	97	0A	0	Store the product (0A <sub>16</sub> ) in address 0063 <sub>16</sub> .
18	0053	96	0A	0	Load the accumulator with the multiplier (00 <sub>16</sub> ) from address 0062 <sub>16</sub> .
19	0055	27	00	1	BEQ. Check the Z flag. Now it's set. Branch to address 0060 <sub>16</sub> .
20	0060	3E	00	1	Halt.

Figure E7-5

## Discussion

The chart that you completed should be similar to the one shown in Figure E7-5. Compare the charts.

The first step we don't see, since it's executed before the Trainer stops at address 0051. Nevertheless, we do see the result of this clear accumulator instruction because the accumulator contains 00. When Step 1 is executed, 00<sub>16</sub> is stored in location 0063<sub>16</sub>. Step 2 brings us to address 0053<sub>16</sub> which loads the accumulator with the multiplier, in this example, 02<sub>16</sub>. The BEQ instruction is next, but the Z flag is cleared so the program continues on the normal route. Next the multiplier is decremented to 01<sub>16</sub> and then stored in location 0062<sub>16</sub>. Now the product (00<sub>16</sub>) is loaded and the multiplicand (05<sub>16</sub>) is added directly. This produces the new product, 05<sub>16</sub>. Now the program encounters the BRA, or branch always instruction and it branches back to address 0051<sub>16</sub>.

Here the new product is stored away and the multiplier is loaded again. It's 01<sub>16</sub> this time, so the program continues on through the BEQ instruction, the multiplier is decremented to 00<sub>16</sub>, and the multiplicand 05<sub>16</sub> is added to the product. The new product (0A<sub>16</sub>) is still in the accumulator. Once again, the BRA instruction loops flow back to address 0051<sub>16</sub> and the product is stored in address 0063<sub>16</sub>.

The multiplier is now loaded and, since it's been decremented to 00<sub>16</sub>, it sets the Z flag. The BEQ instruction checks the Z flag, finds that it's set and branches to the halt instruction at address 0060<sub>16</sub>. Therefore, the program makes two complete passes, before the multiplier becomes 00<sub>16</sub>. On the third pass through, BEQ terminates the program because the Z flag is set.

The multiplier sets the count and determines how many additions will be performed. If the multiplier is changed to 06<sub>16</sub>, the program will make six complete loops, halting on the seventh loop. The BEQ will only be satisfied when the multiplier has been reduced to 00.

As you can see, the branch instructions make decisions based on the contents of the condition code register. It's up to you, however, to choose the correct branch instruction to make the necessary decision.

All branch instructions use relative addressing. In Unit 5, we discussed the method used to calculate the destination address for a branch instruction. However, another shorthand type procedure that's quite

popular with programmers can be used. With this technique, you simply count in hexadecimal. For a forward branch, you begin at  $00_{16}$  and count up to the destination address.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	??	??
1A		
1B		
1C		
1D		
1E		
1F		
20		
21		
22		
23		
24		

Originating address

Destination address

We wish to Branch to here

Figure E7-6

For example, in the program of Figure E7-6, we want to branch from address  $18_{16}$  to address  $24_{16}$ . Recall that the relative address is added to the contents of the program counter. After the BRA instruction and its operand (the relative address) have been fetched, the program counter is pointing the address  $1A_{16}$ . Therefore, we begin our count at address  $1A_{16}$ . Then we count forward in hex as shown in Figure E7-7. When we reach the destination address, the hexadecimal count is the relative address. In this case, it's  $0A_{16}$ , and we insert this operand at address  $19_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	0A	0A
00 1A		
01 1B		
02 1C		
03 1D		
04 1E		
05 1F		
06 20		
07 21		
08 22		
09 23		
0A 24		

Originating Address

Destination Address

Relative Address

Figure E7-7  
Branching forward.



To branch backward in the program, we count down using negative hex numbers. It may sound more difficult, but once you are accustomed to it, you will find it easier to use than the previous method you learned.

For example, in the program shown in Figure E7-8A, we wish to branch back to address  $58_{16}$ . The BRA instruction, at address  $5D_{16}$  is fetched and the program count points to address  $5F_{16}$ . Figure E7-8B shows how we calculate the address for this backward branch. We begin with  $FF_{16}$ , and count down. When we reach the destination address ( $58_{16}$ ), the count at that point is the relative address, in this case  $F9_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56	—	—
57	—	—
58	—	—
59	—	—
5A	—	—
5B	—	—
5C	—	—
5D	20	BRA
5E	??	??
5F		

Program branches to here

Destination Address

Originating Address

(A)

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56	—	—
57	—	—
58	—	—
59	—	—
5A	—	—
5B	—	—
5C	—	—
5D	20	BRA
5E	F9	F9
5F		

Relative address

Destination Address

Originating Address

(B)

Figure 7-8  
Branching back.

Figure E7-9 shows another example of computing the relative address for a larger branch. The branch instruction is at address  $B0_{16}$  and therefore, the origination address is  $B2_{16}$ . We calculate the relative address as shown in Figure E7-9B. Starting with  $FF_{16}$  at address  $B1_{16}$ , we count down to the destination address  $A0_{16}$ . As the count indicates, the relative address to get to  $A0_{16}$  is  $EE_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
A0	Destination	—
A1	Address	—
A2	—	—
A3	—	—
A4	—	—
A5	—	—
A6	—	—
A7	—	—
A8	—	—
A9	—	—
AA	—	—
AB	—	—
AC	—	—
AD	—	—
AE	—	—
AF	—	—
B0	26	BNE
B1	??	??
B2	—	—

(A)

Figure E7-9

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
EE A0	—	—
EF A1	—	—
F0 A2	—	—
F1 A3	—	—
F2 A4	—	—
F3 A5	—	—
F4 A6	—	—
F5 A7	—	—
F6 A8	—	—
F7 A9	—	—
F8 AA	—	—
F9 AB	—	—
FA AC	—	—
FB AD	—	—
FC AE	—	—
FD AF	—	—
FE B0	26	BNE
FF B1	EE	EE
B2	—	—

(B)

In the next section of this experiment, you will write a program that divides by repeated subtraction. You will probably need two branches in this program, a forward branch and a branch back. Use this new technique to calculate the relative addresses for both branches.

## Procedure (Continued)

20. In Unit 5, we discussed a program that divides by repeated subtraction. The flow chart for this program is shown in Figure E7-10. Using this flow chart as a guide and the instructions presented in Figure E7-11, write a program that divides by repeated subtraction. Use a starting address of 0050.

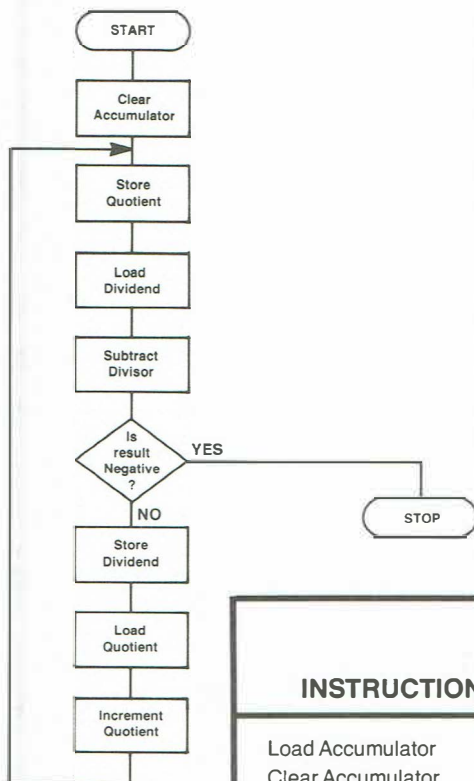


Figure E7-10  
Flow chart for dividing by repeated subtraction.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch if Carry Set	BCS			25	
Branch if Equal Zero	BEQ			27	
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure E7-11  
Instructions to be used.

[illegible]

Figure E7-12

21. Now load the program into the Trainer. Let the dividend be  $0B_{16}$  and the divisor be  $05_{16}$ . Change the program counter to the starting address of your program and single-step through the program, recording the information in the chart of Figure E7-12.
22. Examine the contents of the address that stores the dividend and the quotient. If you followed the flow chart, the address where the dividend is stored should now contain the remainder from the division. Record the contents below.

Quotient \_\_\_\_\_ Remainder \_\_\_\_\_

Address	PC	Dividend	Quotient	Remainder
0000	0000	0000	0000	0000
0001	0001	0000	0000	0000
0002	0002	0000	0000	0000
0003	0003	0000	0000	0000
0004	0004	0000	0000	0000
0005	0005	0000	0000	0000
0006	0006	0000	0000	0000
0007	0007	0000	0000	0000
0008	0008	0000	0000	0000
0009	0009	0000	0000	0000
000A	000A	0000	0000	0000
000B	000B	0000	0000	0000
000C	000C	0000	0000	0000
000D	000D	0000	0000	0000
000E	000E	0000	0000	0000
000F	000F	0000	0000	0000
0010	0010	0000	0000	0000
0011	0011	0000	0000	0000
0012	0012	0000	0000	0000
0013	0013	0000	0000	0000
0014	0014	0000	0000	0000
0015	0015	0000	0000	0000
0016	0016	0000	0000	0000
0017	0017	0000	0000	0000
0018	0018	0000	0000	0000
0019	0019	0000	0000	0000
001A	001A	0000	0000	0000
001B	001B	0000	0000	0000
001C	001C	0000	0000	0000
001D	001D	0000	0000	0000
001E	001E	0000	0000	0000
001F	001F	0000	0000	0000
0020	0020	0000	0000	0000
0021	0021	0000	0000	0000
0022	0022	0000	0000	0000
0023	0023	0000	0000	0000
0024	0024	0000	0000	0000
0025	0025	0000	0000	0000
0026	0026	0000	0000	0000
0027	0027	0000	0000	0000
0028	0028	0000	0000	0000
0029	0029	0000	0000	0000
002A	002A	0000	0000	0000
002B	002B	0000	0000	0000
002C	002C	0000	0000	0000
002D	002D	0000	0000	0000
002E	002E	0000	0000	0000
002F	002F	0000	0000	0000
0030	0030	0000	0000	0000
0031	0031	0000	0000	0000
0032	0032	0000	0000	0000
0033	0033	0000	0000	0000
0034	0034	0000	0000	0000
0035	0035	0000	0000	0000
0036	0036	0000	0000	0000
0037	0037	0000	0000	0000
0038	0038	0000	0000	0000
0039	0039	0000	0000	0000
003A	003A	0000	0000	0000
003B	003B	0000	0000	0000
003C	003C	0000	0000	0000
003D	003D	0000	0000	0000
003E	003E	0000	0000	0000
003F	003F	0000	0000	0000
0040	0040	0000	0000	0000
0041	0041	0000	0000	0000
0042	0042	0000	0000	0000
0043	0043	0000	0000	0000
0044	0044	0000	0000	0000
0045	0045	0000	0000	0000
0046	0046	0000	0000	0000
0047	0047	0000	0000	0000
0048	0048	0000	0000	0000
0049	0049	0000	0000	0000
004A	004A	0000	0000	0000
004B	004B	0000	0000	0000
004C	004C	0000	0000	0000
004D	004D	0000	0000	0000
004E	004E	0000	0000	0000
004F	004F	0000	0000	0000
0050	0050	0000	0000	0000
0051	0051	0000	0000	0000
0052	0052	0000	0000	0000
0053	0053	0000	0000	0000
0054	0054	0000	0000	0000
0055	0055	0000	0000	0000
0056	0056	0000	0000	0000
0057	0057	0000	0000	0000
0058	0058	0000	0000	0000
0059	0059	0000	0000	0000
005A	005A	0000	0000	0000
005B	005B	0000	0000	0000
005C	005C	0000	0000	0000
005D	005D	0000	0000	0000
005E	005E	0000	0000	0000
005F	005F	0000	0000	0000
0060	0060	0000	0000	0000
0061	0061	0000	0000	0000
0062	0062	0000	0000	0000
0063	0063	0000	0000	0000
0064	0064	0000	0000	0000
0065	0065	0000	0000	0000
0066	0066	0000	0000	0000
0067	0067	0000	0000	0000
0068	0068	0000	0000	0000
0069	0069	0000	0000	0000
006A	006A	0000	0000	0000
006B	006B	0000	0000	0000
006C	006C	0000	0000	0000
006D	006D	0000	0000	0000
006E	006E	0000	0000	0000
006F	006F	0000	0000	0000
0070	0070	0000	0000	0000
0071	0071	0000	0000	0000
0072	0072	0000	0000	0000
0073	0073	0000	0000	0000
0074	0074	0000	0000	0000
0075	0075	0000	0000	0000
0076	0076	0000	0000	0000
0077	0077	0000	0000	0000
0078	0078	0000	0000	0000
0079	0079	0000	0000	0000
007A	007A	0000	0000	0000
007B	007B	0000	0000	0000
007C	007C	0000	0000	0000
007D	007D	0000	0000	0000
007E	007E	0000	0000	0000
007F	007F	0000	0000	0000
0080	0080	0000	0000	0000
0081	0081	0000	0000	0000
0082	0082	0000	0000	0000
0083	0083	0000	0000	0000
0084	0084	0000	0000	0000
0085	0085	0000	0000	0000
0086	0086	0000	0000	0000
0087	0087	0000	0000	0000
0088	0088	0000	0000	0000
0089	0089	0000	0000	0000
008A	008A	0000	0000	0000
008B	008B	0000	0000	0000
008C	008C	0000	0000	0000
008D	008D	0000	0000	0000
008E	008E	0000	0000	0000
008F	008F	0000	0000	0000
0090	0090	0000	0000	0000
0091	0091	0000	0000	0000
0092	0092	0000	0000	0000
0093	0093	0000	0000	0000
0094	0094	0000	0000	0000
0095	0095	0000	0000	0000
0096	0096	0000	0000	0000
0097	0097	0000	0000	0000
0098	0098	0000	0000	0000
0099	0099	0000	0000	0000
009A	009A	0000	0000	0000
009B	009B	0000	0000	0000
009C	009C	0000	0000	0000
009D	009D	0000	0000	0000
009E	009E	0000	0000	0000
009F	009F	0000	0000	0000
00A0	00A0	0000	0000	0000
00A1	00A1	0000	0000	0000
00A2	00A2	0000	0000	0000
00A3	00A3	0000	0000	0000
00A4	00A4	0000	0000	0000
00A5	00A5	0000	0000	0000
00A6	00A6	0000	0000	0000
00A7	00A7	0000	0000	0000
00A8	00A8	0000	0000	0000
00A9	00A9	0000	0000	0000
00AA	00AA	0000	0000	0000
00AB	00AB	0000	0000	0000
00AC	00AC	0000	0000	0000
00AD	00AD	0000	0000	0000
00AE	00AE	0000	0000	0000
00AF	00AF	0000	0000	0000
00B0	00B0	0000	0000	0000
00B1	00B1	0000	0000	0000
00B2	00B2	0000	0000	0000
00B3	00B3	0000	0000	0000
00B4	00B4	0000	0000	0000
00B5	00B5	0000	0000	0000
00B6	00B6	0000	0000	0000
00B7	00B7	0000	0000	0000
00B8	00B8	0000	0000	0000
00B9	00B9	0000	0000	0000
00BA	00BA	0000	0000	0000
00BB	00BB	0000	0000	0000
00BC	00BC	0000	0000	0000
00BD	00BD	0000	0000	0000
00BE	00BE	0000	0000	0000
00BF	00BF	0000	0000	0000
00C0	00C0	0000	0000	0000
00C1	00C1	0000	0000	0000
00C2	00C2	0000	0000	0000
00C3	00C3	0000	0000	0000
00C4	00C4	0000	0000	0000
00C5	00C5	0000	0000	0000
00C6	00C6	0000	0000	0000
00C7	00C7	0000	0000	0000
00C8	00C8	0000	0000	0000
00C9	00C9	0000	0000	0000
00CA	00CA	0000	0000	0000
00CB	00CB	0000	0000	0000
00CC	00CC	0000	0000	0000
00CD	00CD	0000	0000	0000
00CE	00CE	0000	0000	0000
00CF	00CF	0000	0000	0000
00D0	00D0	0000	0000	0000
00D1	00D1	0000	0000	0000
00D2	00D2	0000	0000	0000
00D3	00D3	0000	0000	0000
00D4	00D4	0000	0000	0000
00D5	00D5	0000	0000	0000
00D6	00D6	0000	0000	0000
00D7	00D7	0000	0000	0000
00D8	00D8	0000	0000	0000
00D9	00D9	0000	0000	0000
00DA	00DA	0000	0000	0000
00DB	00DB	0000	0000	0000
00DC	00DC	0000	0000	0000
00DD	00DD	0000	0000	0000
00DE	00DE	0000	0000	0000
00DF	00DF	0000	0000	0000
00E0	00E0	0000	0000	0000
00E1	00E1	0000	0000	0000
00E2	00E2	0000	0000	0000
00E3	00E3	0000	0000	0000
00E4	00E4	0000	0000	0000
00E5	00E5	0000	0000	0000
00E6	00E6	0000	0000	0000
00E7	00E7	0000	0000	0000
00E8	00E8	0000	0000	0000
00E9	00E9	0000	0000	0000
00EA	00EA	0000	0000	0000
00EB	00EB	0000	0000	0000
00EC	00EC	0000	0000	000



## Discussion

Now you've written a program that incorporates an unconditional branch and a conditional branch. Hopefully, you calculated the relative addresses using the shorthand technique just discussed. Our program for the divide by repeated subtraction is listed in Figure E7-13. If you followed the flow chart, your program should be similar to this.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
0050	4F	CLRA	Clear the accumulator.
0051	97	→ STA	Store in the quotient which
0052	63	63	is at address location 63 <sub>16</sub> .
0053	96	LDA	Load the accumulator with the
0054	61	61	dividend from location 61 <sub>16</sub> .
0055	90	SUB	Subtract the
0056	62	62	divisor from the dividend.
0057	2B	BMI	If the difference is negative,
0058	07	07	branch down to the Halt instruction.
0059	97	STA	Otherwise, store the difference
005A	61	61	back in location 61 <sub>16</sub> .
005B	96	LDA	Load the accumulator with the
005C	63	63	quotient.
005D	4C	INCA	Increment the quotient by one.
005E	20	BRA	Branch back to instruction
005F	F1	F1	in location 51.
0060	3E	→ HLT	Halt.
0061	0B	0B	Dividend (11 <sub>16</sub> ). (Remainder)
0062	05	05	Divisor (5 <sub>16</sub> ).
0063	—	—	Quotient.

Figure E7-13  
Dividing by repeated subtraction.

Notice that we used the BMI (Branch if Minus) conditional branch instruction. Therefore, the N or negative flag will satisfy the branch when it's set. Figure E7-14 charts our program as we single-stepped through it. Since the program subtracts the divisor from the dividend and stores the difference as the new dividend, at the conclusion of the program the dividend is actually the remainder of the division. When 0B<sub>16</sub> is divided by 05<sub>16</sub>, the quotient should be 02<sub>16</sub> and the remainder 01<sub>16</sub>.

STEP	PROGRAM COUNTER	OPCODE	ACCA	N FLAG	COMMENTS
1	0051	97	00	0	Store the quotient (00 <sub>16</sub> ) at address 0063 <sub>16</sub> .
2	0053	96	00	0	Load the accumulator with the dividend from address 0061 <sub>16</sub> .
3	0055	90	0B ↑ Dividend	0	Subtract the divisor (05 <sub>16</sub> ) at address 0062 <sub>16</sub> from the accumulator.
4	0057	2B	06 ↑ After subtraction	0	BMI. Check the N flag. It's not set so continue.
5	0059	97	06	0	Store the difference (06 <sub>16</sub> ) back in address 0061 <sub>16</sub> .
6	005B	96	06	0	Load the accumulator with the quotient (00 <sub>16</sub> ) at address 0063 <sub>16</sub> .
7	005D	4C	00	0	Increment the quotient.
8	005E	20	01 ↑ Quotient after INC.	0	Branch back to the instruction at address 0051 <sub>16</sub> .
9	0051	97	01	0	Store the quotient (01 <sub>16</sub> ) at address 0063 <sub>16</sub> .
10	0053	96	01	0	Load the accumulator with the dividend (06 <sub>16</sub> ) at address 0061 <sub>16</sub> .
11	0055	90	06 ↑ Dividend Now	0	Subtract the divisor (05 <sub>16</sub> ) at address 0062 <sub>16</sub> from the accumulator.
12	0057	2B	01 ↑ After Subtraction	0	BMI. Check the N flag. It's not set so continue.
13	0059	97	01	0	Store the difference (01 <sub>16</sub> ) back in address 0061 <sub>16</sub> .
14	005B	96	01	0	Load the accumulator with the quotient (01 <sub>16</sub> ) at address 0063 <sub>16</sub> .
15	005D	4C	01	0	Increment the quotient.
16	005E	20	02 ↑ Quotient after INC.	0	Branch back to the instruction at address 0051 <sub>16</sub> .
17	0051	97	02	0	Store the quotient (02 <sub>16</sub> ) at address 0063 <sub>16</sub> .
18	0052	96	02	0	Load the accumulator with the dividend (01 <sub>16</sub> ) at address 0061 <sub>16</sub> .
19	0055	90	01	0	Subtract the divisor (05 <sub>16</sub> ) at address 0062 <sub>16</sub> from the accumulator.
20	0057	2B	FC ↑ Negative Number	1	BMI. Check the N flag. Now it's set so branch to the instruction at address 0060 <sub>16</sub> .
21	0060	3E	FC	1	Halt.

Figure E7-14

So far, we've used the conditional branch instructions only to exit a loop and then halt program execution. However, these branch instructions become even more powerful when they are used to "chain" together different portions of a program. Figure E7-15 shows an example of this chaining effect. The program starts and runs through the first loop until the conditional branch BEQ is satisfied. Then it exits this loop and starts another. When the BEQ condition is satisfied in the second loop, another exit is performed, and another portion of the program is executed.

A strategically placed conditional branch at the end of the program can cause a branch back to the beginning that will repeat the program again and again. In the next portion of this experiment, you will load the BCD-to-binary conversion program that you studied earlier. Then you will step through the program and watch as the Trainer executes each instruction.

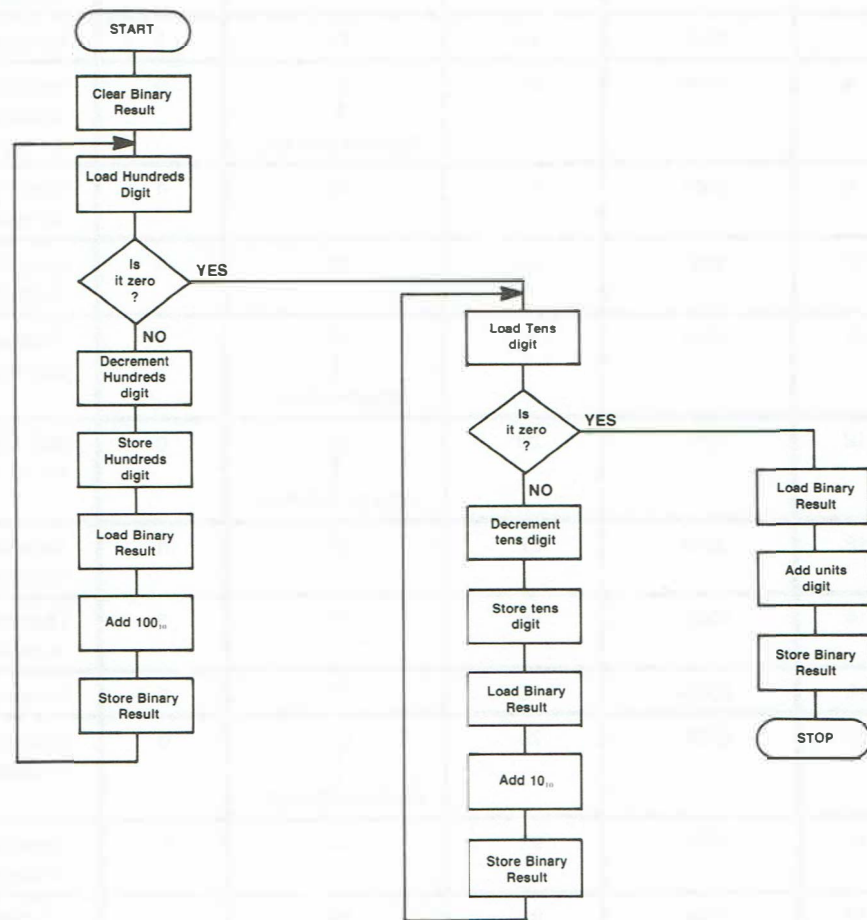


Figure E7-15  
Conditional branches can chain together portions of a program.

## Procedure (Continued)

23. Load the program listed in Figure E7-16 into the Trainer. The BCD number  $117_{10}$  will be converted to binary by this program. Of course, you cannot view the binary contents of the accumulator on your trainer, therefore we will use the hexadecimal equivalent of the binary values during this experiment.

The BEQ instruction is used for the conditional branches in this program. This means that MPU will monitor the \_\_\_\_\_ flag to determine if the condition is set.

24. Now set the program counter to 0050 and single-step through the program recording the information in the chart of Figure E7-17. Notice that, at strategic steps, you should stop and answer questions before you continue. Now single-step through the first 8 steps of the program.

25. What is the hundreds BCD digit at this time? \_\_\_\_\_. The result is now  $64_{16}$ , which is \_\_\_\_\_ in the decimal number system.

Now return to the Trainer and step through the next ten steps of the program.

26. What is the tens BCD digit at this time? \_\_\_\_\_.

The result is now  $6E_{16}$ . This is the equivalent of \_\_\_\_\_ in the decimal number system.

Now return to the Trainer and step through the remainder of the program.

27. Examine address  $007B_{16}$  and record the result below.

\_\_\_\_\_16

Convert this number to its decimal equivalent.

$75_{16} = \text{_____}_{10}$

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	4F	CLRA	Clear the Accumulator.
0051	97	STA	Store00
0052	7B	7B	in location 7B. This clears the binary result.
0053	96	LDA	Load direct into the accumulator
0054	78	78	the hundreds BCD digit.
0055	27	BEQ	If the hundreds digit is zero, branch
0056	0B	0B	forward to the instruction in location 62 <sub>16</sub> .
0057	4A	DECA	Otherwise, decrement the accumulator.
0058	97	STA	Store the result as the new
0059	78	78	hundreds BCD digit.
005A	96	LDA	Load direct into the accumulator
005B	7B	7B	the binary result.
005C	8B	ADD	Add immediate
005D	64	64	100 <sub>10</sub> to the binary result.
005E	97	STA	Store away the new
005F	7B	7B	binary result.
0060	20	BRA	Branch
0061	F1	F1	back to the instruction in location 53 <sub>16</sub> .
0062	96	LDA	Load direct into the accumulator.
0063	79	79	the tens BCD digit.
0064	27	BEQ	If the tens BCD digit is zero, branch
0065	0B	0B	forward to the instruction in location 71 <sub>16</sub> .
0066	4A	DECA	Otherwise, decrement the accumulator.
0067	97	STA	Store the result as the new
0068	79	79	tens BCD digit.
0069	96	LDA	Load direct into the accumulator
006A	7B	7B	the binary result.
006B	8B	ADD	Add immediate
006C	0A	0A	10 <sub>10</sub> to the binary result.
006D	97	STA	Store away the new
006E	7B	7B	binary result.
006F	20	BRA	Branch
0070	F1	F1	back to the instruction in location 62 <sub>16</sub> .
0071	96	LDA	Load direct into the accumulator
0072	7B	7B	the binary result.
0073	9B	ADD	Add direct
0074	7A	7A	the units BCD digit.
0075	97	STA	Store away the new
0076	7B	7B	binary result.
0077	3E	HLT	Halt.
0078	01	01	Hundreds BCD digit.
0079	01	01	Tens BCD digit.
007A	07	07	Units BCD digit.
007B	—	—	Reserved for the binary result.

Figure E7-16  
Program for converting BCD to binary.



STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
Stop! Return to Step 25.					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
Stop! Return to Step 26.					
19					
20					
21					
22					
23					
24					
25					

Figure E7-17

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0051	97	00	1	Store 00 in address 007B <sub>16</sub> . This clears the binary result.
2	0053	96	00	1	Load the accumulator with the Hundreds BCD digit (01 <sub>16</sub> ).
3	0055	27	Hundreds BCD → Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
4	0057	4A	01	0	Decrement the BCD Hundreds Digit.
5	0058	97	New → Hundreds Digit 00	1	Store the new Hundreds Digit (00).
6	005A	96	00	1	Load the accumulator with the Binary Result (00 <sub>16</sub> ).
7	005C	8B	00	1	Add to the binary result 64 <sub>16</sub> .
8	005E	97	Binary → Result Now 64	0	Store away the new binary result.
9	0060	20	64	0	Branch back to address 0053 <sub>16</sub> .
10	0053	96	64	0	Load the accumulator with the Hundreds BCD digit (00).
11	0055	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0062 <sub>16</sub> .
12	0062	96	00	1	Load the accumulator with the tens BCD digit (01 <sub>16</sub> ).
13	0064	27	Tens BCD → Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
14	0066	4A	01	0	Decrement the tens BCD digit (01 <sub>16</sub> ).
15	0067	97	New Tens → Digit 00	1	Store the new tens BCD digit.
16	0069	96	00	1	Load the accumulator with the binary result (64 <sub>16</sub> ).
17	006B	8B	64	0	Add 0A <sub>16</sub> to the binary result.
18	006D	97	New Binary → Result 6E	0	Store away the new binary result.
19	006F	20	6E	0	Branch back to address 0062 <sub>16</sub> .
20	0062	96	6E	0	Load the accumulator with the tens BCD digit (00).
21	0064	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0071 <sub>16</sub> .
22	0071	96	00	1	Load the accumulator with the binary result (6E <sub>16</sub> ).
23	0073	9B	6E	0	Add the units BCD digit (07 <sub>16</sub> ).
24	0075	97	New Binary → Result 75	0	Store the new binary result (75 <sub>16</sub> ).
25	0077	3E	75	0	Halt.

Figure E7-18

Single-stepping through the BCD-to-binary conversion program.

## Discussion

Now you've verified the operation of the BCD-to-binary conversion program. The chart that you completed should match the one shown in Figure E7-18.

Since the BEQ instruction is used for the conditional branches in the program, we monitored the Z flag. In this example, the BCD number  $117_{10}$  was converted to its binary equivalent  $75_{16}$ . This program will convert BCD numbers as high as  $255_{10}$ , to their binary equivalent.

The program isn't as complicated as it might appear. The hundreds and tens BCD digits are used to set a count. Each pass through a loop decrements the BCD digit, or count, and then adds the equivalent hexadecimal positional value for that BCD digit. For example, in the hundreds conversion loop,  $64_{16}$  is added to the binary result for each hundreds BCD digit. Hence, the BCD digit sets the count. Then the count is decremented by one and the program loops back and runs through again. When the count is zero, that BCD digit has been added the correct number of times and the program branches off to another loop. This continues until the program halts.

Stepping through the program, you found that after Step 8, the Trainer had completed one loop through the hundreds BCD portion of the program. The count was  $00_{16}$  and the binary result was  $64_{16}$ , or the binary equivalent of  $100_{10}$ . On the next pass through, the program branches to the tens BCD loop.

The first loop through, the tens BCD portion of the program was completed at Step 18. The binary result was  $6E_{16}$ , which is the equivalent of  $110_{10}$ . The tens BCD digit had been decremented to  $00_{16}$ . Then all that remained was to add the units BCD digit ( $07_{10}$ ) and the conversion process was complete.

You verified the final result by checking the binary result at location  $007B_{16}$ . Here you found the hex number  $75_{16}$ . When you converted this number to its decimal equivalent, you found that  $75_{16}$  equals  $117_{10}$ . Also, if you converted  $75_{16}$  to binary, you would find the number  $0111\ 0101_2$ ; which is the (binary) equivalent of  $117_{10}$ , so the program works.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	4F	CLRA	Clear the accumulator.
0051	97	STA	Store 00
0052	7B	7B	in location 007B <sub>16</sub> . This clears the hundreds digit.
0053	97	STA	Store 00.
0054	7C	7C	in location 007C <sub>16</sub> . This clears the tens digit.
0055	97	STA	Store 00.
0056	7D	7D	in location 007D <sub>16</sub> . This clears the units digit.
0057	96	LDA	Load direct into the accumulator
0058	7A	7A	the binary number to be converted.
0059	80	SUB	Subtract immediate
005A	64	64	100 <sub>16</sub> .
005B	25	BCS	If a borrow occurred, branch
005C	0A	0A	forward to the instruction in location 0066 <sub>16</sub> .
005D	97	STA	Otherwise, store the result of the subtraction
005E	7A	7A	as the new binary number.
005F	96	LDA	Load direct into the accumulator
0060	7B	7B	the hundreds digit of the BCD result.
0061	4C	INCA	Increment the hundreds digit.
0062	97	STA	Store the hundreds digit
0063	7B	7B	back where it came from.
0064	20	BRA	Branch
0065	F1	F1	back to the instruction at address 0057 <sub>16</sub> .
0066	96	LDA	Load direct into the accumulator
0067	7A	7A	the binary number.
0068	80	SUB	Subtract immediate
0069	0A	0A	10 <sub>16</sub> .
006A	25	BCS	If a borrow occurred, branch
006B	09	09	forward to the instruction in location 0075 <sub>16</sub> .
006C	97	STA	Otherwise, store the result of the subtraction
006D	7A	7A	as the new binary number.
006E	96	LDA	Load direct into the accumulator
006F	7C	7C	the tens digit.
0070	4C	INCA	Increment the tens digit.
0071	97	STA	Store the tens digit.
0072	7C	7C	back where it came from.
0073	20	BRA	Branch
0074	F1	F1	back to the instruction at address 0066 <sub>16</sub> .
0075	96	LDA	Load direct into the accumulator
0076	7A	7A	the binary number.
0077	97	STA	Store it in
0078	7D	7D	the units digit.
0079	3E	HLT	Halt.
007A	75	75	Place binary number to be converted at this address.
007B	—	—	Hundreds digit
007C	—	—	Tens digit
007D	—	—	Units digit

} Reserved for  
BCD result.

Figure E7-19

A program with an incorrect relative address.



The most frequent mistake made by programmers when using the branch instructions is the improper computation of the relative address. An improperly coded relative address not only prevents the program from executing properly, but can even wipe out portions of the program. In the next section of this experiment, you will witness the result of an incorrect relative address and the effect it has on the program. In this example, we will use the binary-to-BCD conversion program you studied in Unit 5. The starting address of the program in this experiment is different from the example shown in Unit 5.

### Procedure (Continued)

28. Load the program listed in Figure E7-19 into the Trainer. This program should convert the binary number  $0111\ 0101_2$  ( $75_{16}$ ) into its BCD equivalent. However, one of the relative addresses is **incorrect**. Part of this exercise is to locate the incorrect relative address and correct it.
29. Now set the program counter to 0050 and single-step through the program. Record the results in the chart of Figure E7-20. Notice that we're monitoring the carry (C) flag because the program uses the BCS (Branch if Carry Set) instruction.

STFP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure E7-20

Single-Stepping through the binary-to-BCD conversion program.



30. Examine addresses  $007B_{16}$ ,  $007C_{16}$ , and  $007D_{16}$ ; record the results below.

$007B$  \_\_\_\_ Hundreds BCD Digit

$007C$  \_\_\_\_ Tens BCD Digit

$007D$  \_\_\_\_ Units BCD Digit

Obviously, there is something wrong with the program. Although the hundreds and tens digits are believable, the units digit of 11 is impossible. Remember, a decimal number can only have a units digit of from 0 to  $9_{10}$ .

31. Use the program listing and the chart that you've compiled and locate the error in the program. Then record the address of the instruction below.

HINT: The problem is with the relative address for one of the branch instructions. When one of these addresses is incorrect, the program branches to the wrong address, possibly skipping portions of the program. Therefore, first determine the portions of the program that produced the wrong result and work back until you find the problem.

Address \_\_\_\_ Incorrect Relative Address \_\_\_\_

32. Now calculate the correct relative address (operand) and record it below.

Correct Relative Address \_\_\_\_.

## Discussion

This exercise should have demonstrated the versatility of your Trainer to assist you in “debugging” programs. When you examined addresses 007B<sub>16</sub>, 007C<sub>16</sub>, and 007D<sub>16</sub>, you found these results.

007B   0 1   Hundreds BCD Digit

007C   0 0   Tens BCD Digit

007D   1 1   Units BCD Digit

Obviously, the units BCD digit is incorrect. Since the units digit is wrong, we begin to debug at this portion of the program. This happens to be the least complex section of the program because the binary number is simply loaded into the accumulator and stored in address 007D<sub>16</sub>. Comparing the chart that you compiled against the program listing, we find that this portion of the program seems to be executing correctly.

Therefore, we move back to the tens BCD digit portion of the program. Checking the program listing, we find that the tens BCD portion of the program begins at address 0066<sub>16</sub>. But as the chart in Figure E7-19 shows, when the program is single-stepped the tens BCD digit loop actually starts at address 0067<sub>16</sub>. This is the wrong address. We find the problem when we move back to Step 14 of the chart. This is the BCS (Branch if Carry Set) instruction at address 005B<sub>16</sub>. However, instead of branching to address 0066<sub>16</sub> as the comments column suggests, the program goes to address 0067<sub>16</sub>. Therefore, the relative address at address 005C<sub>16</sub> must be incorrect. When we check this relative address, we find that it should be 09<sub>16</sub>, instead of 0A<sub>16</sub>.

But, how did this incorrect operand affect the program? Following the chart in Figure E7-21, we find that the hundreds BCD portion of the program worked correctly. On the second loop through this portion of the program, the subtraction resulted in a borrow and the C flag was set. Hence, the BCS instruction produced the desired branch.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0051	97	00	0	Store 00 in Hundreds Digit.
2	0053	97	00	0	Store 00 in tens Digit.
3	0055	97	00	0	Store 00 in units Digit.
4	0057	96	00	0	Load the accumulator with the Binary number ( $75_{16}$ ).
5	0059	80	75	0	Subtract $64_{16}$ from accumulator.
6	005B	25	11	0	BCS. Check C flag for borrow. It's clear so continue.
7	005D	97	11	0	Store away the new binary number.
8	005F	96	11	0	Load the accumulator with the Hundreds Digit (00).
9	0061	4C	00	0	Increment the Hundreds Digit.
10	0062	97	01	0	Store the Hundreds Digit.
11	0064	20	01	0	Branch back to address $0057_{16}$ .
12	0057	96	01	0	Load the accumulator with the Binary Number ( $11_{16}$ ).
13	0059	80	11	0	Subtract $64_{16}$ from accumulator. BCS. Check C Flag for borrow.
14	005B	25	AD	1	It's set so branch to address $0066_{16}$ .
	<b>Tens BCD</b>	← Wrong Address			
15	0067	7A	AD	1	What's this?
16	0069	0A	AD	1	
17	006A	25	AD	1	BCS. Check C Flag. It's still set so branch to address $0075_{16}$ .
	<b>Units BCD</b>				
18	0075	96	AD	1	Load the accumulator with the Binary number.
19	0077	97	11	1	Store it in the units Digit.
20	0079	3E	11	1	Halt.

Figure E7-21  
Locating the incorrect relative address.

But, instead of branching to address 0066, where we would have found a load accumulator instruction (96<sub>16</sub>) with an operand of 7A<sub>16</sub>, the program branches to address 0067<sub>16</sub>. The Trainer now interprets the contents of address 0067<sub>16</sub> as an instruction or opcode.

As it turns out, 7A<sub>16</sub> is a valid opcode for the “Decrement Memory” instruction written in the extended addressing mode. These will be discussed in Unit 6. For now, all you need to know is that the MPU executes the 7A<sub>16</sub> instruction and uses the next two bytes of code for the address of the operand for this instruction.

The program execution continues. However, because the LDA instruction at address 0066<sub>16</sub> and the SUB instruction at address 0068<sub>16</sub> are not performed, the results of the program are incorrect.

Therefore, this one incorrect relative address caused the program to skip the tens BCD portion of the program. The tens unit was never subtracted, so it carried over into the units BCD digit. This produced the wrong units digit of 11<sub>10</sub>.

Step 17 finds the program at address 006A<sub>16</sub>. Here, we encounter another BCS conditional branch instruction. The C flag is still set so we branch to address 0075<sub>16</sub>. The program works properly from this point on.

By the way, if you want to run this program again, you must re-enter the binary number to be converted at address 007A<sub>16</sub> because the value at this location is changed each time the program is executed.

## Procedure (Continued)

33. Now change the operand at address  $005C_{16}$  from  $0A_{16}$  to  $09_{16}$ .
34. Also change the number at address  $007A_{16}$  to  $75_{16}$ . This is the number that the program will convert to its BCD equivalent.
35. Reset the program counter to  $0050$  and single-step through the program comparing the program listing with the results that you obtain.
36. Examine the addresses listed below and record the information stored there.

007B \_\_ \_\_      Hundreds BCD Digit

007C \_\_ \_\_      Tens BCD Digit

007D \_\_ \_\_      Units BCD Digit

Is this the correct BCD representation for the number  $75_{16}$ ?

\_\_\_\_\_

## Discussion

When we correct the program by inserting the relative address ( $09_{16}$ ) at address  $007C_{16}$ , we find that it works perfectly. After single-stepping through the program, we examine the BCD digits stored at addresses  $007B_{16}$ ,  $007C_{16}$ , and  $007D_{16}$ . The hundreds digit is  $01_{10}$ , the tens digit is  $01_{10}$ , and the units digit is  $07_{10}$ . Therefore, the BCD equivalent of the binary number  $0111\ 0101_2$  ( $75_{16}$ ) is  $117_{10}$ .



# Experiment 8

## *Additional Instructions*

**OBJECTIVES:**

*To verify the operation of the ADC instruction when used in a multiple-precision addition program.*

*To investigate the hazard of using the ADC instruction when a carry is not desired.*

*To demonstrate your ability to write a multiple-precision subtraction program using the SBC instruction.*

*To demonstrate your ability to write a routine that will multiply any 4-bit binary number times  $16_{10}$  using the ASLA instruction.*

*To verify the operation of a BCD packing program that uses the ASLA instruction.*

*To verify the operation of the DAA instruction when used in a BCD multiple-precision addition program.*

## Introduction

One of the measures of a microprocessor's power is the size of the instruction set. In other words, more instructions generally mean more potential power. You saw the economy that resulted with the addition of branch instructions in the previous experiment. In this experiment, we will examine four additional instructions; the ADC or add with carry, the SBC or subtract with carry, the ASLA or arithmetic shift accumulator left, and the DAA or decimal adjust accumulator.

The discussion in Unit 5 explained the purpose of each instruction. In this experiment, we will restrict our activity to verifying that each instruction works as explained.

In the previous experiment, you examined the condition code registers and determined how the MPU monitors these flag registers to initiate conditional branches. Yet, these condition code registers are also monitored for other instructions. For example, the ADC (add with carry) and SBC (subtract with carry) instructions key on the C or carry flag. If an ADC instruction is executed and the carry flag is set, one is added to the least significant bit in the accumulator. Likewise, if the C flag is set when an SBC instruction is executed, one is subtracted from the least-significant bit of the accumulator. Remember, the C flag represents a "borrow" to the subtract instruction.

In the first portion of this experiment, we will verify the operation of the ADC instruction with a program for multiple precision arithmetic. Then we will examine one of the hazards of using this instruction.

## Material Required

ET-18 Robot Trainer

## Procedure

1. Turn on the Trainer and press the **RESET** key.
2. Load the program listed in Figure E8-1 into the Trainer. This program performs multiple-precision addition of two  $16_{10}$  bit numbers. The augend  $1B93_{16}$  will be added to the addend  $C0EA_{16}$  by this program. Of course, the program can add any numbers that are  $16_{10}$  bits or less.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	01	NOP	No operation
0051	96	LDA	Load the accumulator direct with the
0052	5E	5E	least significant byte of the addend.
0053	9B	ADD	Add direct the
0054	60	60	least significant byte of the augend.
0055	97	STA	Store the result in the
0056	62	62	least significant byte of the sum.
0057	96	LDA	Load the accumulator direct with the
0058	5F	5F	most significant byte of the addend.
0059	99	ADC	Add with carry direct the
005A	61	61	most significant byte of the augend.
005B	97	STA	Store the result in the
005C	63	63	most significant byte of the sum.
005D	3E	HLT	Halt
005E	EA	EA	Least significant byte
005F	C0	C0	Most significant byte
			} addend
0060	93	93	Least significant byte
0061	1B	1B	Most significant byte
			} augend
0062	—	—	Least significant byte
0063	—	—	Most significant byte
			} sum

Figure E8-1

Program for multiple-precision addition.

3. Change the program counter to 0050 and single-step through the program, recording the information in the chart of Figure E8-2. Notice that we are monitoring the carry (C) flag.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure E8-2

4. Examine memory location 0062<sub>16</sub> and 0063<sub>16</sub> and record the sum below.

SUM \_ \_ \_ \_

5. Add the binary numbers below. These numbers are the binary equivalent of the two hex numbers added by the program just executed.

		MSB		LSB
COEA <sub>16</sub>	=	1100	0000	1110 1010
1B93 <sub>16</sub>	=	0001	1011	1001 0011
SUM	=	<hr/>		

Now, convert the binary sum to its hexadecimal equivalent and record below.

SUM \_ \_ \_ \_

Does this match the sum obtained in step 4? \_\_\_\_\_



6. Now load the program of Figure E8-3 into the Trainer. This program simply adds two binary numbers and produces a carry. Hence, it will set the C flag. You will see its purpose in a moment.

Execute the program by pressing the **DO** key and then entering address 0050.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDA	Load the accumulator immediate
0051	EA	EA	with EA <sub>16</sub> .
0052	8B	ADD	Add immediate
0053	93	93	93
0054	3E	HLT	Halt

Figure E8-3

Program adds two numbers and produces carry.

7. Examine the carry (C) condition code register. The C flag is

\_\_\_\_\_

set/reset

**NOTE:** Proceed immediately to the next step of the experiment. **Do not** turn off power to your Trainer before proceeding to Step 8 as this will invalidate this section of the experiment.

8. Enter the program listed in Figure E8-4 into the Trainer. Notice that this is the same multiple-precision addition program previously executed, with the exception that the ADD instruction has been replaced by the ADC instruction, as shown by the shaded section.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	01	NOP	No operation.
0051	96	LDA	Load the accumulator direct with the
0052	5E	5E	least significant byte of the addend.
0053	99	ADC	Add with carry direct the
0054	60	60	least significant byte of the augend.
0055	97	STA	Store the result in the
0056	62	62	least significant byte of the sum.
0057	96	LDA	Load the accumulator direct with the
0058	5F	5F	most significant byte of the addend.
0059	99	ADC	Add with carry direct the
005A	61	61	most significant byte of the augend.
005B	97	STA	Store the result in the
005C	63	63	most significant byte of the sum.
005D	3E	HLT	Halt.
005E	EA	EA	Least significant byte
005F	CO	CO	Most significant byte
			} addend
0060	93	93	Least significant byte
0061	1B	1B	Most significant byte
			} augend
0062	—	—	Least significant byte
0063	—	—	Most significant byte
			} sum

Figure E8-4

Multiple-precision addition program with instruction at address 0053<sub>16</sub> changed.

9. Set the program counter to 0050 and single-step through the program, recording the information in the chart of Figure E8-5.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure E8-5

10. Examine memory locations 0062<sub>16</sub> and 0063<sub>16</sub>. Record the sum below.

SUM \_ \_ \_ \_

Compare this sum to the previous sum recorded in Step 4. Are they the same? \_\_\_\_\_  
yes/no

Why are the sums different? \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

From this demonstration, what conclusion can you draw concerning the use of the ADC instruction? \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

## Discussion

In Steps 1 through 3 of this experiment, you loaded a multiple-precision addition program similar to the one you studied in Unit 4. Single-stepping through the program, you witnessed the operation of the ADC instruction. The chart you compiled should be similar to the chart in Figure E8-6. When you checked memory locations  $0062_{16}$  and  $0063_{16}$ , you found the LSB and MSB respectively of the  $16_{10}$ -bit sum. The sum should have been  $DC7D_{16}$ .

In Step 5 you added the binary equivalents of the hex numbers,  $COEA_{16}$  and  $1B93_{16}$ . The sum was the binary equivalent of the sum produced by the program, as shown below.

		MSB		LSB
			1	
$COEA_{16}$	=	1100	0000	1110 1010
$1B93_{16}$	=	0001	1011	1001 0011
SUM	=	1101	1100	0111 1101

As you noticed, a carry is generated by the addition of the least significant bytes of the two numbers. When you were single-stepping through the program, you observed this carry because the C flag was set. The addition of the most significant bytes did not produce a carry. Therefore, the carry flag was cleared.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0051	96	Random	Random	Load the accumulator with the LSB of Addend ( $EA_{16}$ ).
2	0053	9B	EA	Random	Add the LSB of the Augend ( $93_{16}$ ).
3	0055	97	7D	1	Store result in LSB of sum.
4	0057	96	7D	1	Load the accumulator with the MSB of the Addend ( $CO_{16}$ ).
5	0059	99	C0	1	Add with carry the MSB of the Augend ( $1B_{16}$ ).
6	005B	97	DC	0	Store result in MSB of Sum.
7	005D	3E	DC	0	Halt.

Figure E8-6

When you converted the binary number to hexadecimal, you found that the sum was the same as that produced by the program.

1101 1100 0111 1101

D C 7 D

In Step 6, you loaded a simple program that added the numbers  $EA_{16}$  and  $93_{16}$ . Of course, the addition generated a carry, as you witnessed when you checked the C flag and found it set.

In Step 8, you loaded another multiple-precision addition program into the Trainer. The only difference between this program and the previous multiple-precision addition program was that the first add instruction was the ADC (add with carry), rather than the ADD. Then you single-stepped through the program and completed the chart of Figure E8-5. Your chart should be similar to the one shown in Figure E8-7.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0051	96	Random	1	Load the accumulator with the LSB of Addend ( $EA_{16}$ ).
2	0053	99	EA	1	Add with carry the LSB of the Augend ( $93_{16}$ ).
3	0055	97	7E	1	Store result in LSB of sum. Load the accumulator with the MSB of Addend ( $CO_{16}$ ).
4	0057	96	7E	1	
5	0059	99	C0	1	Add with carry the MSB of the Augend ( $1B_{16}$ ).
6	005B	97	DC	0	Store result in MSB of sum.
7	005D	3E	DC	0	Halt.

Figure E8-7

Single-stepping through the multiple-precision addition program where both add instructions are ADC.



When you examined the sum at addresses  $0062_{16}$  and  $0063_{16}$ , you found  $DC7E_{16}$ . The correct sum, as you verified earlier, should have been  $DC7D_{16}$ . If you checked the chart compiled while single-stepping through the program, the reason for this incorrect answer should have been evident. Because the short program in Figure E8-3 generated a carry, the carry flag was set even before the multiple precision program was executed. Therefore, when the Trainer executed the first ADC instruction, it automatically added the carry ( $1_2$ ) to the sum of the least significant bytes. Hence, the result  $7E$  was one greater than the correct sum of  $7D$ .

From this demonstration you should have reached the conclusion that the ADC instruction should not be used unless you need to add a generated carry and you know which instruction generates the carry. You must remember that the C flag is only reset by an arithmetic operation that doesn't produce a **carry** or a **borrow**. For example, in the program that worked properly, we used the simple ADD instruction for the first addition. Naturally, this instruction ignores the condition of the C flag, so it doesn't matter if it's set or reset. This is a simple way of playing it safe. The second addition used the ADC instruction because we wanted any carry from the least significant byte to be reflected in the most significant byte.

The SBC (subtract with carry) instruction is similar to the ADC instruction because it also monitors the C flag to indicate a borrow. In the next section of this experiment, you will write a program that uses the SBC instruction for multiple-precision subtraction of  $16_{10}$ -bit numbers.

## Procedure (Continued)

11. Write a program that will perform multiple-precision subtraction of two  $16_{10}$ -bit (2-byte) numbers. The following guidelines define the problem. Use  $0050_{16}$  as your starting address.
  - A. The program must subtract a  $16_{10}$ -bit subtrahend from a  $16_{10}$ -bit minuend and store the difference in memory.
  - B. Use the direct addressing mode.
  - C. Select the opcodes from the instruction listing in Figure E8-8.
12. Now load the program. Enter  $9721_{16}$  in the locations reserved for the minuend and  $7581_{16}$  in the locations reserved for the subtrahend.
13. Single-step through the program and observe its operation. Examine the locations where the difference is stored and record the 2-byte difference below.

DIFFERENCE \_\_\_\_\_

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Add with Carry	ADC	89	99		
Subtract with Carry	SBC	82	92		
Arithmetic Shift					
Accumulator Left	ASLA				48
Decimal Adjust					
Accumulator	DAA				19
Halt	HLT				3E

Figure E8-8  
Instructions.

## Discussion

If you made a flow chart of the problem, it probably looks like the one shown in Figure E8-9. Your program should be similar to the solution shown in Figure E8-10. After stepping through the program on the Trainer, the difference of the subtraction should have been  $21A0_{16}$ . If you didn't obtain this answer, go back and recheck your program.

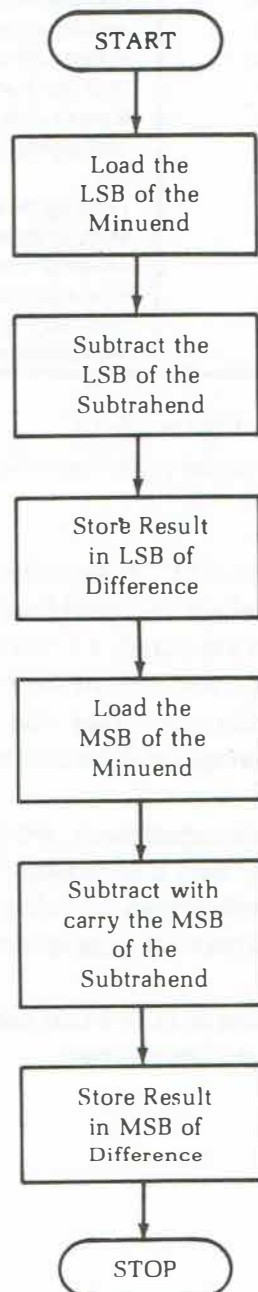


Figure E8-9

Flow chart for multiple-precision subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load accumulator direct with
0051	5D	5D	least significant byte of minuend
0052	90	SUB	Subtract direct
0053	5F	5F	least significant byte of subtrahend
0054	97	STA	Store result in
0055	61	61	least significant byte of difference
0056	96	LDA	Load accumulator direct with
0057	5E	5E	most significant byte of minuend
0058	92	SBC	Subtract with carry
0059	60	60	most significant byte of the subtrahend
005A	97	STA	Store result in
005B	62	62	most significant byte of difference
005C	3E	HLT	Halt
005D	21	21	Least significant byte
005E	97	97	Most significant byte
005F	81	81	Least significant byte
0060	75	75	Most significant byte
0061	—	—	Least significant byte
0062	—	—	Most significant byte

Figure E8-10

Program for multiple-precision subtraction.

You may have used the SBC instruction for the first subtraction. If you did, this might explain the problem; because if the C flag is set when this instruction is executed, a 1 will be borrowed from the difference. Therefore, your answer would have been 1 less than the correct answer, or  $21A0_{16}$ . If the carry flag was cleared before you executed the program, the result would still be correct.

In the next section of this experiment, we will examine the ASLA (arithmetic shift accumulator left) instruction. You will also write a simple program that uses this instruction to multiply any  $4_{10}$ -bit number by  $16_{10}$ . This simple routine will prove its usefulness later.

Recall from the discussion in Unit 5 that each ASLA operation multiplies the contents of the accumulator by two.

## Procedure (Continued)

14. Using the instructions listed in Figure E8-8 write a program that uses the ASLA instruction to multiply any  $4_{10}$ -bit number by  $16_{10}$ . Use  $0050_{16}$  as your starting address.
15. Enter your program into the Trainer and then have your program multiply  $0F_{16}$  ( $15_{10}$ ) by  $16_{10}$ . Record the product below.

$$0F_{16} \times 16_{10} = \underline{\hspace{2cm}}_{16}.$$

16. Convert the product obtained to its decimal equivalent.

$$\text{Decimal equivalent } \underline{\hspace{2cm}}_{10}.$$

Now check your result by multiplying  $15_{10}$  times  $16_{10}$ .

$$15_{10} \times 16_{10} = \underline{\hspace{2cm}}_{10}.$$

17. In this program, the multiplier is determined by the number of ASLA instructions. How many ASLA instructions are required to produce a multiplier of  $16_{10}$ ?           .



## Discussion

The program for this simple routine is shown in Figure E8-11. Notice that it uses four ASLA instructions to produce the required multiplier of  $16_{10}$ . If your program worked properly, the final product should have been  $F0_{16}$ . Converting this number to its decimal equivalent, we find that  $F0_{16}$  equals  $240_{10}$ . When we multiplied  $15_{10}$  times  $16_{10}$ , we also found the product was  $240_{10}$ . Therefore, the program works.

When using the ASLA instruction, each shift left multiplies the number in the accumulator by two. Therefore, a single shift multiplies a number by two, the next shift multiplies the original number by four; and so on.

Another use for the ASLA instruction is to pack two BCD digits into a single byte. This “packing” can result in a significant savings of memory if many BCD numbers are used. Let’s verify the operation of the BCD packing program that was presented in Unit 5.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load the accumulator with the
0051	59	59	4-bit multiplicand
0052	48	ASLA	} Shift the accumulator four places to the left multiplying the multiplicand by $16_{10}$ .
0053	48	ASLA	
0054	48	ASLA	
0055	48	ASLA	
0056	97	STA	Store the product
0057	5A	5A	at this location
0058	3E	HLT	Halt
0059	5F	5F	4-bit multiplicand
005A	—	—	Product

Figure E8-11

Program that uses the ASLA instruction to multiply a 4-bit number times  $16_{10}$ .

## Procedure (Continued)

18. Enter the BCD packing program listed in Figure E8-12 into the Trainer. The unpacked BCD numbers are  $09_{10}$  and  $03_{10}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	01	NOP	Do nothing.
0051	96	LDA	Load into the accumulator direct
0052	5D	5D	the unpacked most significant BCD digit.
0053	48	ASLA	} Shift it four places to the left.
0054	48	ASLA	
0055	48	ASLA	
0056	48	ASLA	
0057	9B	ADD	Add the
0058	5E	5E	unpacked least significant BCD digit.
0059	97	STA	Store the result
005A	5C	5C	in the packed BCD number.
005B	3E	HLT	Halt.
005C	00	00	Packed BCD number.
005D	09	09	Unpacked most significant BCD digit.
005E	03	03	Unpacked least significant BCD digit.

Figure E8-12

Program to pack two BCD digits into a single byte.

19. Set the program counter to 0050 and single-step through the program, recording the information below. Where it is indicated, convert the hexadecimal contents of the accumulator to the binary equivalent.

Program Count	Opcode	ACCA	Binary Equivalent
0051	96	Random	Random
0053	48	_____	_____
0054	48	_____	_____
0055	48	_____	_____
0056	48	_____	_____
0057	9B	_____	_____
0059	97	_____	_____
005B	3E	HALT	

20. Examine the packed BCD number at address  $005C_{16}$  and record it below.

Packed BCD Number \_\_\_\_\_

## Discussion

As you can see, the BCD packing program is very simple. Nevertheless, simple routines such as this can be combined in many programs, easing the task of programming. Most programmers either commit these general purpose routines to memory or file them away for future reference.

The results you obtained by stepping through the program should be similar to those shown below.

<u>Program Count</u>	<u>Opcode</u>	<u>ACCA</u>	<u>Binary Equivalent</u>
0051	96	Random	Random
0053	48	09	0000 1001
0054	48	12	0001 0010 After 1st shift
0055	48	24	0010 0100 After 2nd shift
0056	48	48	0100 1000 After 3rd shift
0057	9B	90	1001 0000 After 4th shift
0059	97	93	1001 0011
005B	3E		

As the listing shows, the most significant BCD digit (09<sub>10</sub>) is loaded into the accumulator. Four ASLA shifts take place, moving this digit progressively to the left. Following these four shifts, the most significant BCD digit is properly positioned. Now the program simply adds the least significant BCD (03<sub>10</sub>) to the contents of the accumulator and then stores the sum. Checking the address of the packed BCD number, we find 93<sub>10</sub>.

When BCD numbers are added, we encounter yet another problem. Often, the sum is the correct BCD number. But, just as frequently, it isn't. In Unit 4, the reason for this inconsistency was discussed. However, your Trainer has an instruction, called the "Decimal Adjust Accumulator" (DAA), that can correct the sum of BCD numbers, producing the desired result.

In the next portion of this experiment, we will demonstrate the need for the DAA instruction by first adding two BCD numbers without using the DAA instruction. Then we will check the sum. Next, we will correct the program by inserting DAA instructions and again examine the BCD sum.

## Procedure (Continued)

21. Load the program listed in Figure E8-13 into your Trainer. This program adds the BCD numbers  $3792_{10}$  and  $5482_{10}$ , storing the sum in address  $0061_{16}$  and  $0062_{16}$ .
22. RESET the Trainer and execute the program by first pressing the 1 and then the **DO** key and entering address 0050.
23. Again, press the **RESET** key and then examine the sum stored at address  $0061_{16}$  and  $0062_{16}$ . The most significant byte of the sum is at address  $0061_{16}$  and the least significant byte is at address  $0062_{16}$ . Record the sum below.

SUM \_\_\_\_\_

Is this the correct BCD sum for the addition of the numbers  $3792_{10}$  and  $5482_{10}$ ? \_\_\_\_\_.

yes/no

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDA	Load the accumulator direct with
0051	5E	5E	the least significant byte of addend.
0052	9B	ADD	Add direct
0053	60	60	the least significant byte of augend.
0054	97	STA	Store the result in
0055	62	62	the least significant byte of BCD sum.
0056	96	LDA	Load the accumulator direct with
0057	5D	5D	the most significant byte of addend.
0058	99	ADC	Add with carry
0059	5F	5F	the most significant byte of augend.
005A	97	STA	Store the result in
005B	61	61	the most significant byte of BCD sum.
005C	3E	HLT	Halt.
005D	37	37	Most significant byte
005E	92	92	Least significant byte } BCD Addend
005F	54	54	Most significant byte
0060	82	82	Least significant byte } BCD Augend
0061	—	—	Most significant byte
0062	—	—	Least significant byte } BCD Sum

Figure E8-13

Incorrect program for multiple-precision addition of BCD numbers.

24. Now load the corrected multiple-precision BCD addition program listed in Figure E8-14 into your Trainer. Notice that the only changes between this program and the previous program are the additions of the NOP instructions and the two DAA instructions following the addition operations.
25. Change the program counter to 0050 and single-step through the program, recording the information below.

STEP 1

<u>PROGRAM COUNT</u>	<u>OPCODE</u>
----------------------	---------------

STEP 2

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>
----------------------	---------------	-------------

STEP 3

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>	<u>C FLAG</u>
----------------------	---------------	-------------	---------------

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	01	NOP	Do nothing.
0051	96	LDA	Load the accumulator direct with the
0052	61	61	least significant byte of addend.
0053	9B	ADD	Add direct
0054	63	63	the least significant byte of augend.
0055	19	DAA	Decimal adjust the sum to BCD.
0056	97	STA	Store the result in the
0057	65	65	least significant byte of BCD sum.
0058	96	LDA	Load the accumulator direct with the
0059	60	60	most significant byte of addend.
005A	99	ADC	Add with carry the
005B	62	62	most significant byte of augend.
005C	19	DAA	Decimal adjust the sum to BCD.
005D	97	STA	Store the result in the
005E	64	64	most significant byte of BCD sum.
005F	3E	HLT	Halt.
0060	37	37	Most significant byte } BCD Addend
0061	92	92	Least significant byte }
0062	54	54	Most significant byte } BCD Augend
0063	82	82	Least significant byte }
0064	—	—	Most significant byte } BCD Sum
0065	—	—	Least significant byte }

Figure E8-14

Program for adding multiple-precision BCD numbers.



The sum of the addition of the least significant bytes is now in the accumulator. Is this the correct BCD sum for the numbers  $92_{10}$  and  $82_{10}$ ? \_\_\_\_\_  
yes/no

When the DAA instruction (opcode 19) is executed, will this number be corrected? \_\_\_\_\_  
yes/no

STEP 4

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>	<u>C FLAG</u>
----------------------	---------------	-------------	---------------

As you can see, the DAA instruction did correct the left-most digit by adding  $60_{16}$  to the sum. Since the result  $14_{10}$  appears to be a legitimate BCD number, how did the MPU know it was not the valid BCD sum? \_\_\_\_\_  
\_\_\_\_\_

STEP 5

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>	<u>C FLAG</u>
----------------------	---------------	-------------	---------------

STEP 6

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>	<u>C FLAG</u>
----------------------	---------------	-------------	---------------

STEP 7

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>	<u>C FLAG</u>
----------------------	---------------	-------------	---------------

It's obvious that this number ( $8C_{16}$ ) is not the BCD sum of  $37_{10}$  and  $54_{10}$ . What number will the MPU add to  $8C_{16}$  to produce the desired BCD sum? \_\_\_\_\_.

STEP 8

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>	<u>C FLAG</u>
----------------------	---------------	-------------	---------------

STEP 9

<u>PROGRAM COUNT</u>	<u>OPCODE</u>	<u>ACCA</u>
----------------------	---------------	-------------

26. Now examine the BCD sum at addresses  $0064_{16}$  and  $0065_{16}$  and record below.

SUM \_\_\_\_\_<sub>10</sub>.

## Discussion

When you executed the first program to add BCD numbers, it was obvious that the sum 8C14 was not the correct BCD number. The answer should have been 9274<sub>10</sub>. Naturally, the MPU considered these BCD numbers as hexadecimal numbers, hence, the hexadecimal sum.

However, when the program was modified by the addition of DAA (decimal adjust accumulator) instructions after each addition operation, the result was the correct BCD number. As you stepped through the program you saw the DAA instruction in operation.

At Step 3, the BCD numbers 92<sub>10</sub> and 82<sub>10</sub> had been added and the accumulator was supposedly storing the sum 14<sub>10</sub>. A carry was generated by the setting of the C flag. However, the sum was not correct. Instead of 14<sub>10</sub>, the sum should have been 174<sub>10</sub>. To the MPU, the addition looked something like this.

	1001	0010 <sub>2</sub>	= 92 <sub>16</sub>
C FLAG	1000	0010 <sub>2</sub>	= 82 <sub>16</sub>
<hr/>			
1 Carry	0001	0100 <sub>2</sub>	114 <sub>16</sub>

If we ignore the carry, the sum 14<sub>16</sub> appears to be a legitimate BCD number. Nevertheless, the sum would be incorrect. Taking the carry flag into consideration, remember it's just an extension of the accumulator, we find the sum is 114<sub>16</sub>. In hex, this is the correct sum of the two numbers.

When using the DAA instruction, if the sum of either the 4 MSB's or 4 LSB's of the result exceeds 1001<sub>2</sub>, 0110<sub>2</sub> is automatically added to this value to "adjust" the result.

In Step 4, the DAA instruction had been executed and, as you witnessed, the number  $14_{16}$  had been adjusted to the correct BCD sum of  $74_{10}$ . The carry flag was set, indicating that the sum of the two left-most 4-bit binary numbers was larger than  $1001_2$  ( $9_{16}$ ). Actually, it was  $1\ 0001_2$ . When the DAA instruction was executed, the MPU followed the conversion rules and adjusted the sum by adding  $60_{16}$  as shown below.

Carry					Carry	
1	0001	0100 <sub>2</sub>	=	1	14 <sub>16</sub>	
	0110	0000 <sub>2</sub>	=		60 <sub>16</sub>	
<hr/>						
1	0111	0100 <sub>2</sub>	=	1	74 <sub>16</sub>	

The result is  $74_{16}$  with a carry of  $1_{16}$ . This is the correct BCD sum for the two BCD numbers. If we include the carry, the result is  $174_{10}$  which is indeed the decimal sum of  $92_{10}$  and  $82_{10}$ . However, this exceeds the capacity of our storage locations, since they're only 8-bits long, so the carry is carried forward to the addition of the most significant bytes of the numbers in the next step.

As you continued single-stepping through the program, the most significant bytes were loaded and added with the ADC instruction. At step 7, the sum of this addition was in the accumulator. It was obvious that the sum  $8C_{16}$  wasn't a BCD number. To adjust this number to the correct BCD sum,  $06_{16}$  was added by the DAA instruction. The BCD adjusted sum  $92_{10}$  was the result.

In the final step of the experiment, you verified program operation by examining the BCD sum at locations  $0064_{16}$  and  $0065_{16}$ . Here you should have found the sum  $9274_{10}$ .

1. The first step in the process of identifying a problem is to recognize that a problem exists. This is often done by comparing actual performance with desired performance. If there is a significant difference, a problem is identified.

2. The second step is to define the problem. This involves determining the scope of the problem, the causes of the problem, and the consequences of the problem. This step is often done by gathering data and analyzing it.

3. The third step is to develop a solution. This involves brainstorming possible solutions, evaluating the pros and cons of each solution, and selecting the best solution. This step is often done by a team of people who are familiar with the problem.

4. The fourth step is to implement the solution. This involves putting the solution into action and monitoring its progress. This step is often done by a team of people who are responsible for the solution.

5. The fifth step is to evaluate the solution. This involves determining whether the solution has been successful in solving the problem. This step is often done by a team of people who are familiar with the problem.

# Experiment 9

## *New Addressing Modes*



**OBJECTIVES:**

*To gain experience using the instruction set and registers of the MPU.*

*To demonstrate the indexed addressing mode.*

**Introduction**

In Unit 6, you learned that the MPU has two new addressing modes called extended and indexed addressing. Either of these addressing modes can be used to reach operands anywhere in memory. By contrast, the direct addressing mode can be used only when the operand is in the first  $256_{10}$  bytes of memory.

**Material Required**

ET-18 Robot Trainer

## Procedure

- Figure E9-1 shows a program for adding a list of numbers. Because the numbers are in addresses higher than  $00FF_{16}$ , the extended addressing mode is used. Load this program into the Trainer and verify that you have loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0100	4F	CLRA	Clear accumulator A
0101	BB	ADDA	Add the first number
0102	01	01	which is at this
0103	20	20	address.
0104	BB	ADDA	Add the second number.
0105	01	01	
0106	21	21	
0107	BB	ADDA	Add the third number.
0108	01	01	
0109	22	22	
010A	BB	ADDA	
010B	01	01	
010C	23	23	
010D	BB	ADDA	
010E	01	01	
010F	24	24	
0110	BB	ADDA	
0111	01	01	
0112	25	25	
0113	BB	ADDA	
0114	01	01	Continue until all numbers are added.
0115	26	26	
0116	BB	ADDA	
0117	01	01	
0118	27	27	
0119	BB	ADDA	
011A	01	01	
011B	28	28	
011C	BB	ADDA	
011D	01	01	
011E	29	29	
011F	3E	WAI	Stop.
0120	01	01	First number.
0121	02	02	Second number.
0122	03	03	Third number.
0123	04	04	
0124	05	05	•
0125	06	06	•
0126	07	07	•
0127	08	08	
0128	09	09	
0129	0A	0A	Tenth number.

Figure E9-1

Adding a list of numbers using extended addressing.

2. Execute the program using the single-step mode. The first instruction sets the contents of accumulator A to \_\_\_\_\_.
3. Examine the program counter and accumulator A after each instruction is executed. Each time an ADDA extended instruction is executed, the program counter is advanced \_\_\_\_\_ locations.
4. Examine the contents of accumulator A after the final instruction is executed. The number in accumulator A is \_\_\_\_\_.
5. Refer to your instruction set summary card. How many MPU cycles are required to execute this program? \_\_\_\_\_.

## Discussion

The program adds the ten numbers giving the sum  $55_{10}$  or  $37_{16}$ . It requires 51 MPU cycles. Notice that the program itself takes up  $32_{10}$  bytes of memory. The data (the ten numbers) use another  $10_{10}$  bytes.

As you can see, the address of each number added in the program is contained in the two bytes following the ADDA instruction written in the extended mode. For example, after the first ADDA instruction, we have the address of the first operand,  $0120_2$ . The program executes the instructions in order and each instruction specifies the address of the next operand in the list being added.

A repetitive program like this one is an excellent candidate for indexed addressing. Let's see how the same job can be done using indexed addressing.

## Procedure (Continued)

6. Figure E9-2 shows a program for adding the same list of numbers. However it uses indexed addressing. Load this program into the Trainer and verify that you have loaded it correctly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0130	4F	CLRA	Clear accumulator A.
0131	CE	LDX#	Load the index register immediately
0132	01	01	with the address of
0133	20	20	the first number in the list.
0134	AB	→ADDA, X	Add to accumulator A indexed
0135	00	00	with 00 offset.
0136	08	INX	Increment index register.
0137	8C	CPX#	Compare the index register immediately
0138	01	01	with one greater than the address
0139	2A	2A	of the last number in the list.
013A	26	BNE	If there is no match,
013B	F8	F8	branch back to here.
013C	3E	WAI	Otherwise, halt.

Figure E9-2

Adding the list of numbers using indexed addressing.

7. Execute the program using the single-step mode. After each step, record the contents of the program counter, accumulator A, and the index register in Figure E 9-3.
8. Compare the programs of Figures E9-1 and E9-2. Which requires fewer instructions?
9. Refer to the instruction set summary card. How many machine cycles are required to execute the program shown in Figure E9-1 \_\_\_\_\_? Compare this with the number of machine cycles required for the program in Figure E9-2.

STEP NUMBER	CONTENTS AFTER EACH STEP		
	PC	ACCA	INDEX
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			

Figure E9-3  
Record values here.



## Discussion

You probably noticed that this program is significantly shorter than the previous program and yet it produces the same results. This is because indexed addressing uses the contents of the index register to hold the address of the instruction operand. Therefore, it is not necessary to have an address written into the program for each number in the list. It is only a matter of changing the contents of the index register. Let's look at the program to see what happened.

The first instruction clears the accumulator to prepare for program execution. Remember, never assume that the contents of the accumulator are zero.

The next instruction, LDX, loads the starting address of the list into the index register. Now we are ready to begin to add.

The ADDA, X instruction adds the contents of the address pointed to by the index register to the accumulator.

The Increment X instruction changes the address in the index register to the next address in the list. Of course, we don't want to add forever. The CPX# instruction compares the contents of the index register to the address of the last number in the list.

If the last number in the list has not been reached, the BNE is taken and another addition occurs. Furthermore, the contents of the index register are incremented once again and compared to the final address of the list. Once the final address of the list is reached, the BNE is not taken and the program is complete.

This example illustrates that when a repetitive task is to be done, indexed addressing can save many bytes of memory. In many cases, indexed addressing requires more MPU cycles and therefore, a longer time to execute. Generally, time is of little importance compared to saving a substantial number of memory bytes.

Let's look at some other ways that indexed addressing is used.

## Procedure (Continued)

10. Write a program that will clear memory locations  $0120_{16}$  through  $01A0_{16}$ . It should use indexed addressing. The program should reside in the lower RAM addresses.
11. When you are sure your program is correct, load it into your Trainer. Verify that you loaded it correctly; then execute it using the DO command.
12. Examine memory locations  $0120_{16}$  through  $01A0_{16}$ . Each should be cleared. Examine locations below  $0120_{16}$  and above  $01A0_{16}$ . These locations should not be cleared.
13. Debug your program if necessary and repeat Steps 11 and 12 until the desired results are obtained.

## Discussion

Our solution to the problem is shown in Figure E9-4. Your solution may be similar or quite different. If it achieves the proper result and requires about the same number of bytes, then it is perfectly acceptable.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0100	CE	LDX #	Load index register immediately with
0101	01	01	the address of the
0102	20	20	first location to be cleared.
0103	6F	CLR, X	Clear the location whose
0104	00	00	address is indicated by the index register.
0105	08	INX	Increment the index register.
0106	8C	CPX #	Compare the number in the index
0107	01	01	register with one greater than
0108	A1	A1	the address of the last location to be cleared.
0109	26	BNE	If there is no match
010A	F8	F8	branch back to here.
010B	3E	WAI	Otherwise, stop.

Figure E9-4

Program for clearing addresses  $0120_{16}$  through  $01A0_{16}$ .

You can see that this program uses the same basic format as the preceding program.

We still have not demonstrated the full power of indexed addressing because we have not yet used the offset capability. Let's look at how the offset capability can be used. Figure E9-5 shows three tables. The first two tables contain signed numbers, the third is initially cleared. The entries in the first two tables are to be added and the resulting sums are to be placed in the third table. That is, the first entry in Table 1 is to be added to the first entry in Table 2. The resulting sum is to be stored as the first entry of Table 3. The second entry in Table 1 is to be added to the second entry in Table 2, forming the second entry in Table 3; etc.

## Procedure (Continued)

14. Enter the data shown in Figure E9-5 into the indicated addresses.

TABLE 1		TABLE 2		TABLE 3	
ADDRESS	CONTENTS	ADDRESS	CONTENTS	ADDRESS	CONTENTS
0100	06	0110	FA	0150	00
0101	0F	0111	01	0151	00
0102	06	0112	1A	0152	00
0103	20	0113	10	0153	00
0104	2F	0114	11	0154	00
0105	00	0115	50	0155	00
0106	2F	0116	31	0156	00
0107	61	0117	0F	0157	00
0108	3E	0118	42	0158	00
0109	4F	0119	41	0159	00
010A	91	011A	0F	015A	00
010B	9F	011B	11	015B	00
010C	C0	011C	00	015C	00
010D	84	011D	4C	015D	00
010E	70	011E	70	015E	00
010F	E1	011F	0F	015F	00

Figure E9-5  
Three tables.

15. Write a program that will solve the problem described above.
16. Enter the program into the Trainer and execute it.

17. Examine addresses 0150<sub>16</sub> through 015F<sub>16</sub> to verify that the program performed properly.
18. If necessary, debug your program and try again.

## Discussion

The solution to the problem is shown in Figure E9-6.

The first instruction LDX# places the starting address of Table 1 into the index register. Next, the contents of memory at this address are loaded into the accumulator.

The add indexed instruction, ADDA, X, at 0055 in the program uses the contents of the index register and an offset of 10<sub>16</sub> to determine the address of the operand in Table 2.

Finally, the STAA, X instruction uses an offset of 50 hex to determine the address in Table 3.

By changing the contents of the index register with an INX instruction, the address in each table can be changed.

The program is still written in the same basic format as the preceding programs.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	CE	LDX#	Load index register with address
0051	01	01	of first entry
0052	00	00	in Table 1.
0053	A6	LDAA, X	Load entry from Table 1 into
0054	00	00	accumulator A.
0055	AB	ADDA, X	Add the corresponding entry from
0056	10	10	Table 2.
0057	A7	STAA, X	Store the result in the
0058	50	50	corresponding location in Table 3.
0059	08	INX	Increment the index register.
005A	8C	CPX#	Compare the number in the index
005B	01	01	register with one greater
005C	10	10	than the address of the last entry in Table 1.
005D	26	BNE	If there is no match,
005E	F4	F4	branch to here.
005F	3E	WAI	Otherwise, stop.

Figure E9-6  
Program for adding two tables.

# Experiment 10

## *Arithmetic Operations*



**OBJECTIVES:**

*To gain practice using the instruction set and registers of the MPU.*

*To demonstrate a fast method of performing multiplication.*

*To demonstrate a multiple-precision arithmetic.*

*To demonstrate an algorithm for finding the square root of a number.*

*To gain experience writing programs.*

## **Introduction**

In Unit 6, you were exposed to the full architecture and instruction set of the 6808 microprocessor. In this experiment, you will use some of the new-found capabilities of the microprocessor to solve some simple problems.

Mathematical operations make excellent programming examples and at the same time illustrate useful procedures. For these reasons, the programs developed in this experiment are concerned with arithmetic operations.

In an earlier unit, you learned that a computer can multiply by repeated addition. However, this is a very slow method of multiplication when large numbers are used.

A much faster method of multiplying involves a shifting-and-adding process. To illustrate the procedure, consider the longhand method of multiplying two 4-bit binary numbers. The procedure looks like this.

$1101_2$	$\leftarrow$	Multiplicand	$\rightarrow$	$13_{10}$
<u><math>1011_2</math></u>	$\leftarrow$	Multiplier	$\rightarrow$	<u><math>11_{10}</math></u>
$1101$				$13$
$1101$				<u><math>13</math></u>
$0000$				$143_{10}$
<u><math>1101</math></u>				
$10001111_2$	$\leftarrow$	Product	$\nearrow$	

The decimal equivalents are shown for comparison purposes. The product is formed by shifting and adding the multiplicand. Put in computer terms, the procedure goes like this:

1. Clear the product.
2. Examine the multiplier. If it is 0, stop. Otherwise, go to 3.
3. Examine the LSB of the multiplier. If it is 1, add the multiplicand to the product then go to 4. If it is a 0, go to 4 without adding.
4. Shift the multiplicand to the left.
5. Shift the multiplier to the right so that the next bit becomes the LSB.
6. Go to 2.

## Material Required

ET-18 Robot Trainer

## Procedure

1. Write a program of any length that will perform multiplication in the manner indicated. Here are some guidelines:
  - A. You may use any of the instructions discussed up to this point.
  - B. To keep the program simple, only unsigned 4-bit binary numbers are to be used for the multiplier and the multiplicand.
  - C. The final product should be in accumulator A when the multiplication is finished.
  - D. The multiplier may be destroyed during the multiplication process.
  - E. Assume that the multiplier and multiplicand are initially in memory. That is, you should load them into memory along with the program.
2. Try to write the program before you read further. If after 30 minutes, you feel you are not making progress, go on to Step 3.
3. If you feel you need help, read over the following hints and then write the program.
  - A. The product should be formed in accumulator A.
  - B. The first step is to clear the product.
  - C. The multiplicand is shifted and added to accumulator A. Accumulator B is a good place to hold the multiplicand during this process.
  - D. You can test the multiplier for zero while it is still in memory by using the TST instruction followed by the BEQ instruction.
  - E. A good way to test the LSB of the multiplier is to shift the multiplier one bit to the right into the carry flag and then test the carry flag with a BCC instruction.
4. Once your program is written, load it into the Trainer and run it. Verify that it works for several different values of multipliers and multiplicands. Debug your program as necessary.

## Discussion

The real test of your program is “Does it work?” If it works, then you have successfully completed this part of the experiment. One solution to the problem is shown in Figure E10-1. Compare your program with this one. If you could not write a successful program, study the explanation of this program carefully. Note how each phase of the operation is handled.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	4F	CLRA	Set the product to 0.
0051	D6	LDAB	Load accumulator B with the
0052	62	62	multiplicand.
0053	7D	TST	Test
0054	00	00	the
0055	63	63	multiplier.
0056	27	BEQ	If it is 0, branch to the
0057	09	09	wait instruction.
0058	74	LSR	Shift the LSB of the
0059	00	00	multiplier to the
005A	63	63	right into the carry flag.
005B	24	BCC	If the carry flag is cleared,
005C	01	01	skip the next instruction.
005D	1B	ABA	Add the multiplicand to the product.
005E	58	ASLB	Shift the multiplicand to the left.
005F	20	BRA	Branch back and go through again.
0060	F2	F2	
0061	3E	WAI	Wait.
0062	05	Multiplicand	
0063	03	Multiplier	

Figure E10-1

Multiplying by shifting and adding.

In our solution, the first instruction, CLRA, clears accumulator A. This is necessary because the product is obtained through addition. Any value that may be in the accumulator will be added to the product resulting in an incorrect answer. Therefore, the accumulator must be cleared.

The next instruction, LDAB, loads the multiplicand into the B accumulator.

The TST instruction checks to see if the multiplier is zero by comparing the contents of memory location 0063 with 00. If the multiplier is zero, the BEQ is taken and the program is finished.

If the multiplier is not zero, the next five instructions in the program perform the actual multiplication. The LSR instruction uses the LSB of the multiplier to either set or clear the carry flag by shifting the contents of memory location 0063.

If the carry flag is set then the contents of the B accumulator are added to the A accumulator. This is the same as a multiply by the LSB of the multiplier.

If the carry flag is clear, no addition of the accumulators occurs.

Now the contents of the B accumulator are shifted to the left by the ASLB instruction. This is the shift that must take place before the next add in the multiplication process. You can see this more easily if you look at the longhand example on Page 12-135.

Now the branch is taken until the multiplier becomes zero. The multiplier will become zero because the LSR instruction at address 0058 eventually shifts all of the 1's out of memory location 0063.

Obviously, this simple program has some serious drawbacks. The chief one is that the product cannot exceed eight bits. Fortunately, the basic procedure can be expanded so that much larger numbers can be handled.

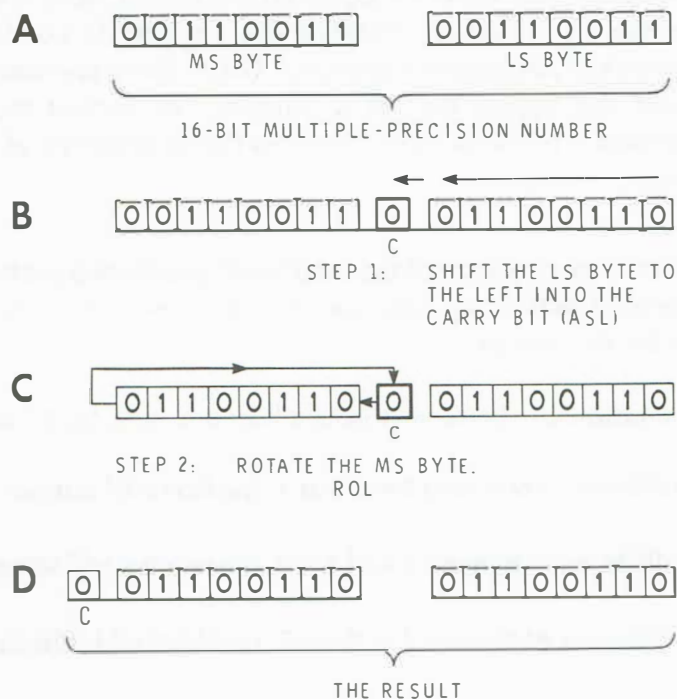
The solution is to use two bytes for the product. This will allow products up to  $65,535_{10}$ . In the next example, the multiplier will be restricted to eight bits. However, the multiplicand can have up to 16 bits (two bytes) as long as the product does not exceed  $65,535_{10}$ . In an earlier unit, you learned that multiple-precision numbers can be added by a 2-step operation. The least significant (LS) byte of one number is added to the LS byte of the other. Then, the MS byte is added **with carry** to the MS byte of the other. Keep this in mind as you write your program.

The procedure for shifting a multiple-precision value will also come in handy. To shift a 2-byte number to the left, a 2-step procedure like that shown in Figure E10-2 can be used. First, the LS byte is shifted one place to the left into the carry bit by using the ASL instruction. Next the MS byte is rotated to the left. The result is that the 16-bit number has been shifted one bit to the left.



## Procedure (Continued)

5. Write a program that will multiply a double-precision multiplicand times an 8-bit multiplier. Assume that the double-precision product is to be stored in memory locations 0040<sub>16</sub> and 0041<sub>16</sub>. The double-precision multiplicand is initially in addresses 0042<sub>16</sub> and 0043<sub>16</sub>. The 8-bit multiplier is in address 0044<sub>16</sub>.
6. Once again, you should try to write this program. If after 30 minutes or so you are not making progress, read the hints given in Step 7.
7. Read over the following hints (if necessary) and try again.
  - A. Initially clear both bytes of the product.
  - B. Test the multiplier for zero exactly as you did in the previous program.
  - C. Test the LSB of the multiplier as you did in the previous program.
  - D. When you are adding the multiplicand to the product, use the multiple-precision add technique.
  - E. When you shift the multiplicand to the left, use the technique shown in Figure E10-2.



**Figure E10-2**  
Shifting a multiple-precision number.

8. Once your program is written, load it into the Trainer and verify that it works properly. Debug the program as necessary.

## Discussion

There are dozens of ways in which this program could be written. If your program produces proper results, then you have been successful. One solution to the problem is shown in Figure E10-3. Compare your program with this one. If you were unsuccessful in writing a program, study Figure E10-3 very carefully until you understand the procedures involved.

This program is essentially the same as the first program in this experiment. Some differences are that the product is now held in two memory addresses rather than in the accumulators, the multiplicand is held in two memory locations, and computations occur in both the A and B accumulators. All of these changes were necessary because the product and the multiplicand are 16-bit numbers.

The most significant change in the program is the addition of the ADCB instruction. This instruction assures that any carry generated by multiplication in the LS byte of the multiplicand is taken into account in the final result. Other than this one instruction, the multiplication loop operates in the same way as the one previously described.

Another problem that makes a good programming exercise is finding the square root of a number. Writing the program is not too difficult once you develop the proper algorithm. While there are many different ways to find the square root of a number, the easiest method from the programmer's point of view involves the subtraction of successive odd integers.

This method works because of the relationship between perfect squares. The first several perfect squares are  $0^2 = 0$ ,  $1^2 = 1$ ,  $2^2 = 4$ ,  $3^2 = 9$ ,  $4^2 = 16$ ,  $5^2 = 25$ , etc. Notice:

The relationship between the numbers 0, 1, 4, 9, 16, 25, etc.

The difference between 0 and 1 is 1, the first odd integer.

The difference between 1 and 4 is 3, the second odd integer.

The difference between 4 and 9 is 5, the third odd integer; etc.

This relationship gives us a simple method of finding the exact square root of perfect squares and of approximating the square root of non-perfect squares.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0040	—	—	Product (LS byte)
0041	—	—	Product (MS byte)
0042	—	—	Multiplicand (LS byte)
0043	—	—	Multiplicand (MS byte)
0044	—	—	Multiplier
*	*	*	Instructions start at address 0050.
0050	7F	CLR	Clear the product.
0051	00	00	
0052	40	40	
0053	7F	CLR	
0054	00	00	
0055	41	41	
0056	7D	→TST	Test the multiplier.
0057	00	00	
0058	44	44	
0059	27	BEQ	If the multiplier is 0, branch to
005A	19	19	the WAI instruction.
005B	74	LSR	Otherwise, shift the right most
005C	00	00	bit of the multiplier into
005D	44	44	the C flag.
005E	24	→BCC	If the C flag is 0 branch to
005F	0C	0C	here.
0060	96	LDAA	Otherwise, load the LS byte of
0061	40	40	the product into accumulator A.
0062	9B	ADDA	Then add the LS byte of the
0063	42	42	multiplicand.
0064	D6	LDAB	Load the MS byte of the product
0065	41	41	into accumulator B.
0066	D9	ADCB	Add (with carry) the MS byte of the
0067	43	43	multiplicand.
0068	97	STAA	Store the contents of accumulator A
0069	40	40	as the LS byte of the product.
006A	D7	STAB	Store the contents of accumulator B
006B	41	41	as the MS byte of the product.
006C	78	→ASL	Shift the LS byte of the
006D	00	00	multiplicand to the left.
006E	42	42	
006F	79	ROL	Rotate the MS byte of the
0070	00	00	multiplicand to the left.
0071	43	43	
0072	20	→BRA	Repeat the process.
0073	E2	E2	
0074	3E	WAI	Stop.

Figure E10-3

Program for multiplying a double-precision multiplicand  
by an 8-bit multiplier.

The procedure for finding the square root of a number looks like this:

1. Subtract successive odd integers (1, 3, 5, 7, 9, etc.) from the number until the number is reduced to 0 or a negative value.
2. Count the number of subtractions required. The count is the exact square root of the number if the number was a perfect square. The count is the approximate square root if the number was not a perfect square.

For example, you could find the square root of  $49_{10}$  as follows:

49	Original Number.
<u>- 1</u>	Subtract the first odd integer.
48	
<u>- 3</u>	Subtract the second odd integer.
45	
<u>- 5</u>	Subtract the third odd integer.
40	
<u>- 7</u>	Subtract the fourth odd integer.
33	
<u>- 9</u>	Subtract the fifth odd integer.
24	
<u>- 11</u>	Subtract the sixth odd integer.
13	
<u>- 13</u>	Subtract the seventh odd integer.
0	Stop subtracting because the original number has been reduced to 0.

Then, simply count the number of subtractions.

Since 7 subtractions were required, the square root of 49 is 7.

### Procedure (Continued)

9. With pencil and paper, use the above algorithm to find the square root of  $81_{10}$ . Does the answer give the exact square? \_\_\_\_\_. Was the result of the final subtraction 0? \_\_\_\_\_.
10. With pencil and paper, use the above algorithm to find the square root of  $119_{10}$ . How many subtractions are required to reduce the number to a negative value. Does this count approximate the square root of  $119_{10}$ ? \_\_\_\_\_.

11. Write a program that uses the above algorithm to find or approximate the square root of any unsigned 8-bit number.
12. Load your program into the Trainer and run it. Verify that it works for several different values.

## Discussion

Our solution to the problem is shown in Figure E10-4. The number is loaded into accumulator A, where it will be gradually reduced to a negative value. The odd integer is maintained in accumulator B. Each new odd integer is formed by incrementing twice. The SBA instruction is used to subtract the odd integer from the number. The BCS instruction is used to determine when the number goes negative (a borrow occurs at that point). You could have used the BMI instruction but this would limit the original number to a value below  $+128_{10}$ . A few bytes are saved by not maintaining a separate count of the number of subtractions. Instead, the final odd integer value is converted to the count. This is possible because of the relationship between the odd integer value and the number of subtractions. As the program is written, the final odd integer is always one more than twice the number of subtractions. By shifting the final odd integer to the right, the correct count is created.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDAA	Load the number that is at
0051	5F	5F	this address into accumulator A.
0052	C6	LDAB#	Load accumulator B with the
0053	01	01	first odd integer.
0054	10	SBA	Subtract the odd integer from the number.
0055	25	BCS	If the carry is set, branch
0056	04	04	to here.
0057	5C	INCB	Otherwise, form the next higher odd
0058	5C	INCB	integer by incrementing B twice.
0059	20	BRA	Branch back
005A	F9	F9	to here.
005B	54	LSRB	Shift the odd integer to the right.
005C	D7	STAB	Store the answer at
005D	60	60	this address.
005E	3E	WAI	Wait.
005F	—	Number	Number to be operated upon.
0060	—	Answer	Final answer appears here.

Figure E10-4  
Square root subroutine.



Of course, any square root program that is limited to numbers below  $256_{10}$  is of limited use. However, this same technique can be applied to multiple-precision numbers. Figure E10-5 shows a program that can find or approximate the square root of numbers up to  $16,385_{10}$ . Before you study this program, try to write your own program to do this.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	96	LDAA	Load accumulator A with the
0051	6A	6A	LS byte of the number.
0052	D6	LDAB	Load accumulator B with the
0053	69	69	MS byte of the number.
0054	7F	CLR	Clear
0055	00	00	the odd
0056	6B	6B	integer.
0057	7C	→ INC	Increment.
0058	00	00	the odd
0059	6B	6B	integer.
005A	90	SUBA	Subtract the odd
005B	6B	6B	integer from the LS byte of the number.
005C	C2	SBCB#	Take care of any borrow
005D	00	00	from the MS byte of the number.
005E	25	BCS	If the carry is set, branch
005F	05	05	to here.
0060	7C	→ INC	Otherwise, form the next
0061	00	00	higher odd integer by
0062	6B	6B	incrementing
0063	20	BRA	and branching
0064	F2	→ F2	to here.
0065	74	→ LSR	Convert the odd integer to
0066	00	00	the answer by shifting
0067	6B	6B	right.
0068	3E	WAI	Stop.
0069	—	Number (MS)	Number to be
006A	—	Number (LS)	operated upon.
006B	—	Odd integer	Form the odd integer and the answer here.

Figure E10-5  
Routine for finding the square root of a  
double precision number.

# Experiment 11

## *Stack Operations*

**OBJECTIVES:**

*To demonstrate the stack operations that occur automatically.*

*To demonstrate ways that the programmer can use the stack.*

*To demonstrate the break-point capability of the Trainer.*

## **Introduction**

As you learned in Unit 6, the stack can be quite useful in a number of operations. It can be used as a temporary storage location for groups or lists of numbers that you are using in a program.

In addition, the MPU uses the stack when executing certain instructions. Among these are: BSR, JMP, JSR, RTI, RTS, SWI, WAI, PULA, PULB, PSHA, PS HB, DES, TXS, and TSX.

In this experiment, you will examine some ways of manipulating the stack. These experiment programs are limited in scope. Just remember, the purpose of this experiment is to show you how the stack works. Once you understand stack operations and the limitations on the use of the stack, you will be able to apply it to your programs.

## **Material Required**

ET-18 Robot Trainer

## Procedure

- Figure E11-1 shows a program for setting the contents of the MPU registers to a known state. The first five bytes of code in this program disable the clock to the auxiliary circuits of the robot. This is done because the robot monitor routine uses a number of stacking routines. If you try to use the circuits, the robot may become "confused."

Examine the remainder of the program, starting at address 0055, and determine the hex contents of the following registers after the WAI instruction is executed.

Condition Code Register \_\_\_\_\_  
 Accumulator B \_\_\_\_\_  
 Accumulator A \_\_\_\_\_  
 Index Register \_\_\_\_\_  
 Program Counter \_\_\_\_\_

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
*0050	86	LDAA#	Load accumulator A with
*0051	90	90	90.
*0052	B7	STAA	Store the contents of accumulator
*0053	C3	C3	A at address
*0054	00	00	C300.
0055	8E	LDS#	Load the number
0056	00	00	0070 into the
0057	70	70	stack pointer.
0058	CE	LDX#	Load the number
0059	EE	EE	EEDD into the
005A	DD	DD	index register.
005B	C6	LDAB#	Load the number
005C	BB	BB	BB into accumulator B.
005D	86	LDAA#	Load the number
005E	AA	AA	AA into accumulator A.
005F	36	PSHA	Push AA onto the stack.
0060	86	LDAA#	Load the number
0061	CC	CC	CC into accumulator A.
0062	06	TAP	Transfer CC to the condition code register.
0063	32	PULA	Pull AA from the stack.
0064	3E	WAI	Wait.

Figure E11-1

This routine sets the contents of all MPU registers to known values.

\*Clock disable routine.

2. Load the program into the Trainer and verify that you loaded it properly.
3. Execute the program using the DO command.
4. Examine the following memory locations and record their hex contents.

<u>Address</u>	<u>Contents</u>	<u>Register</u>
006A	—	—
006B	—	—
006C	—	—
006D	—	—
006E	—	—
006F	—	—
0070	—	—

5. Identify the register from which these numbers came.
6. Try to examine the contents of ACCA, ACCB, PC, SP, and INDEX register. Do their contents agree with the number loaded there?

## Discussion

When the WAI instruction is executed, the contents of the MPU registers are pushed onto the stack. Remember, the contents of these registers are pushed onto the stack in a specific sequence. The contents of the program counter are pushed first. After that, the contents of the index register, accumulator A, accumulator B, and condition codes are placed into the stack as shown here.

<u>Address</u>	<u>Contents</u>	<u>Where it came from</u>
006A	CC	Condition Codes
006B	BB	Accumulator B
006C	AA	Accumulator A
006D	EE	Index Register (high byte)
006E	DD	Index Register (low byte)
006F	00	Program Counter (high byte)
0070	65	Program Counter (low byte)



The LDS# instruction places 0070 into the stack pointer register. This sets the top of the stack at memory location 0070. This location is high enough in memory so that the stack will not extend into the area in memory that contains the program. When using the stack, always make sure that you have enough room in memory to accommodate the stack contents.

Naturally, when this program was written, it was planned that the A accumulator would contain AA and condition code register would contain CC after the execution of the WAI instruction. It would seem a simple matter to load the condition code register from accumulator A with a TAP instruction and then load the A accumulator with AA. However, many times, the apparent solution to a problem will not work.

Look at the program beginning at address 005D. AA is loaded into the A accumulator and then pushed onto the stack. Shortly, you will see why this is necessary. Next, CC is loaded into the A accumulator and transferred to the condition code register with a TAP instruction. Now that the condition code register contains the desired value, the AA is pulled from the stack and placed in the A accumulator.

If you look at the instruction set on Page 6-50 of your text, you will see that the PULA instruction does not affect the contents of the condition code register. Therefore, a PULA can be used to place AA into the A accumulator. If you try to accomplish this with a LDAA instruction, you alter the contents of the condition code register.

When writing a program, always consult the instruction set. Be sure that any instruction you use does exactly what you want it to do.

When you tried to examine the contents of ACCA, ACCB, SP, etc., you found that their contents did not agree with what was loaded. The reason for this **apparent** error is that the Trainer does not actually examine the contents of these registers. Instead, it examines what is placed in the stack by the WAI instruction. However, when the Trainer is reset, the monitor program assumes that the stack starts at address 0FD7. Since our program moved the location of the stack, we can not use the ACCA, ACCB, PC, SP, CC, or INDEX commands after changing the stack pointer and then resetting the Trainer.

This demonstrates how the MPU uses the stack. A similar operation occurs for the SWI instruction or when a hardware interrupt occurs. Of course, the programmer can also use the stack.

## Procedure (Continued)

7. Figure E11-2 shows a program that will clear memory locations 0041 through 005F. It then transfers a list of numbers to these addresses. The numbers come from addresses 0151 through 016F.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0060	86	LDAA#	} Clock Disable Routine.
0061	90	90	
0062	B7	STAA	
0063	C3	C3	
0064	00	00	} Load the index register with highest address to be cleared.
0065	CE	LDX#	
0066	00	00	} Clear it.
0067	1F	1F	
0068	6F	CLR, X	} Offset address value.
0069	40	40	
006A	09	DEX	} Decrement index register to next lower address.
006B	26	BNE	
006C	FB	FB	} Finished? If not, go back and clear the indicated address.
006D	08	INX	
006E	8E	LDS#	} Set index register to first entry in new list.
006F	01	01	
0070	50	50	} Set the stack pointer to one less than the first entry in the old list.
0071	32	PULA	
0072	A7	STAA, X	} Pull the entry from the old list.
0073	40	40	
0074	08	INX	} Store it in the new list.
0075	8C	CPX#	
0076	00	00	} Offset address value.
0077	20	20	
0078	26	BNE	} Increment index register to next entry in list.
0079	F7	F7	
007A	3E	WAI	Finished?
			If not, go back and pull next entry.
			Otherwise, wait.

Figure E11-2

Program for demonstrating stack operations and break points.

8. Load this program into the Trainer and verify that you loaded it properly.
9. At address 0151 through 016F, load the numbers 01 through 1F<sub>16</sub>, respectively.
10. Execute the program using the DO command.
11. Examine addresses 0041 through 005F. They should contain the numbers 01 through 1F, respectively.

## Discussion

This illustrates how the stack can be used in conjunction with indexing to move a list of numbers.

When you execute this program by using the DO command, everything happens so fast that it is impossible to see intermediate results. Of course, you could use the single-step mode and examine the result produced by every single instruction. But in many programs, this is a long, tedious process. Therefore, the Trainer provides another way to examine programs. It allows us to set four different breakpoints in our program. The Trainer will execute instructions at its normal speed until it reaches one of these breakpoints. At that point, the Trainer will stop with the address and opcode of the next instruction displayed. While the Trainer is stopped, you can examine and change the contents of any register or memory location. When you are ready to resume, you depress the return (RTI) key and the Trainer executes instructions at its normal speed until the next breakpoint or a WAI instruction is encountered.

## Procedure (Continued)

12. Verify that the program is still in memory.
13. Depress the **RESET** key. Do not depress **RESET** again as you perform the following steps. To do so will erase any breakpoints that you set.
14. Refer to the program listing in Figure E11-2. Let's assume you wish to stop and examine memory and the MPU registers just before the BNE instruction at address 006B is executed.
15. Depress the 1 and then the BR key. The display should be \_ \_ \_ Br. The Trainer is now ready to accept the first breakpoint address. Enter the address at which the Trainer is to stop: 006B. The breakpoint is now entered.
16. Without hitting **RESET**, depress the DO key. Enter the address of the first instruction in the program: 0060.
17. Immediately, the display will show the address 006B and opcode 26 at which the breakpoint occurred.
18. Without hitting **RESET**, examine the contents of the index register. It should now read 001E.
19. Depress the **EXAM** key and examine address 005F. It should now be cleared.
20. Notice that you can examine the contents of any MPU register or memory location from this breakpoint mode.
21. When you are ready for the program to resume, depress the **RTI** key once. Again, the display will read 006B26 because the MPU is back at the same breakpoint on the second pass through the first loop.
22. Examine the index register again. It should now read 001D. Examine location 005E and verify that it has been cleared.
23. The loop will be repeated 31<sub>10</sub> times. On the 32<sup>nd</sup> pass, the program will escape the loop.

24. Before you go further, set a second breakpoint at the INX instruction. Do this by depressing the BR key and entering the address of the instruction (006D).
25. Depress the RTI key again. Notice that the program is still stopping at the first breakpoint. It will continue to do so until it escapes the first loop.
26. You have now pushed the RTI key three times. Repeatedly push the RTI key until the display changes to 006D 08. The RTI key should have been depressed a total of  $32_{10}$  times, counting the first three times.
27. The program is now waiting at the second breakpoint.
28. To demonstrate a point, let's set two additional breakpoints.
29. Depress the BR key and enter address 006E. This sets the third breakpoint at the LDS# instruction.
30. Depress the BR key again and enter address 0078. This sets the fourth breakpoint at the last BNE instruction.
31. The Trainer will accept only four breakpoints. We have now reached this limit. Depress the BR key again in an attempt to enter a fifth breakpoint. Notice that the word "FULL!" appears on the display.
32. Depress the RTI key so that the Trainer resumes program execution. It should stop at the third breakpoint.
33. Depress the RTI key again. The program should stop at the fourth breakpoint. Notice that the program is again in a loop. On each pass through the loop, the program will stop at this fourth breakpoint.
34. Analyze the operation of the program by examining the pertinent registers and memory locations on each pass through the loop.



## Discussion

The breakpoint capability of the Trainer can be a powerful help to you in writing, analyzing, and debugging a program. It allows you to stop at four distinct points in the program. Here are some tips to remember when you are using this capability:

1. A maximum of four breakpoints can be used.
2. These may be entered all at once or during a previous breakpoint pause.
3. The **RESET** key erases all breakpoints.
4. The contents of the address at which the breakpoint is set must be an opcode.

## Experiment 12

### *Subroutines*

**OBJECTIVES:**

*To demonstrate the use of subroutines.*

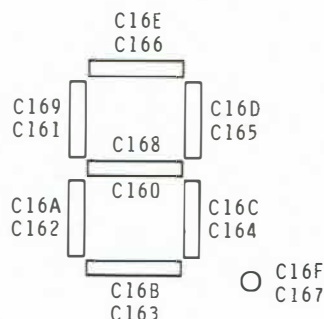
*To demonstrate that the monitor program of the ET-18 Robot Trainer contains some useful subroutines that can be called when needed.*

*To gain experience writing programs.*

## Introduction

Most of the subroutines that you will develop and use in this experiment deal with lighting the displays on the Trainer. For this reason, we will begin by discussing how the displays are accessed.

The ET-18 Robot Trainer has six hexadecimal displays. Each display contains eight light-emitting diodes (LEDs) arranged as shown in Figure E12-1. Each LED is given two addresses. The addresses for the left-most display are shown. To light a particular LED we simply store an odd number at the proper address. An odd number is used because the LED responds to a 1 in bit 0 of the byte that is stored. To turn an LED off, we store an even number at the proper address. The following procedure will demonstrate this.



**Figure E12-1**

Addresses of the various segments in the left LED display.

## Materials Required

ET-18 Robot Trainer

## Procedure

1. Write a program that will halt after storing an odd number (such as 01) at address C167<sub>16</sub>.
2. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the decimal point of the left-most display lit.
3. Notice that the LED remains lit until it is deliberately turned off.

## Discussion

To form characters, the LED's in the display must be turned on in combination. For example, to form the letter "A", the segments at addresses C162, C161, C166, C165, C164, and C160 must be turned on. Notice that the fourth bit, A<sub>3</sub>, may be either 1 or 0. The Trainer's display decoder ignores that bit. Therefore, address C169<sub>16</sub> and C168<sub>16</sub> are the same.

## Procedure (Continued)

4. Write a program that will halt after storing an odd number (such as 01) at the six addresses listed above.
5. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the letter A in the left-most display.

## Discussion

Your program probably took this form:

```
LDAA  #      01
STAA  C162
STAA  C161
STAA  C166
      .
      .
      .
WAI
```

While this approach works, the program would have to be rewritten for each new character. What is needed is a program that will form many characters. One approach is to store characters as 8-bit character bytes. Since there are eight LED's in each display, each bit of the character byte can be assigned to a different LED segment. Figure E12-2A shows how each bit in a character byte is assigned to each segment of the display. To light a corresponding LED, the proper bit in the character byte must be 1. For example, Figure E12-2B shows the character byte for the letter A. To form this letter, all display segments except C163 and C167 must be lit. Therefore, a 1 is placed in the character byte at all bits except the two that correspond to these addresses.

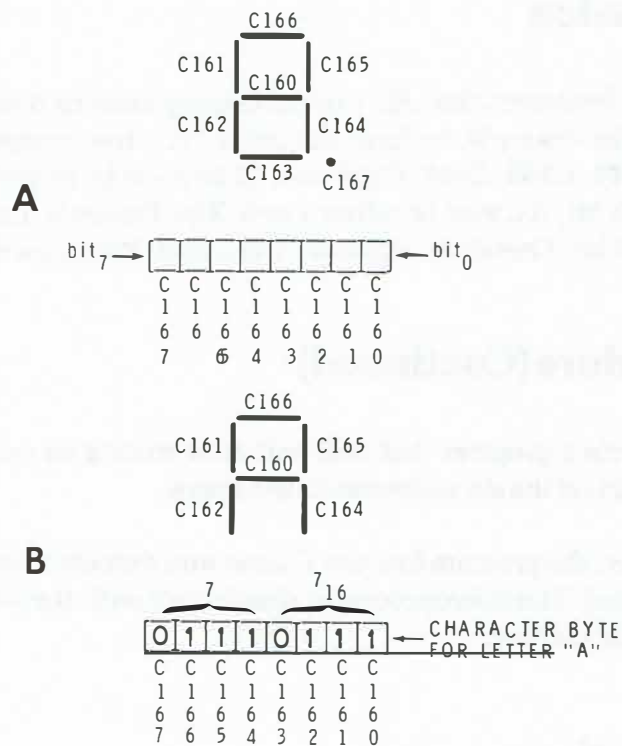


Figure E12-2  
Assigning the bits of the character byte.



The display responds only to bit 0 of the character byte. To make each segment bit appear in turn at bit 0, the character byte must be shifted to the right. After each shift, the contents of the character byte must be stored at the address whose corresponding bit is now at bit 0. The procedure is:

1. Store the contents of the character byte at C160<sub>16</sub>.
2. Shift the character byte to the right.
3. Store it at C161<sub>16</sub>.
4. Shift it to the right again.
5. Store it at C162<sub>16</sub>.

Etc.

A program that will do this is shown in Figure E12-3.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDAA#	Load accumulator A immediate with the character byte. Load the index register immediate with the address. of the left display. Store the character byte at the address indicated by the index register. Shift the character bit to the right. Advance index register to the address of the next segment. Compare index register with one greater than the address of the last segment. If no match occurs branch back to here. Otherwise, stop.
0051	77	77	
0052	CE	LDX#	
0053	C1	C1	
0054	60	60	
0055	A7	STAA, X	
0056	00	00	
0057	44	LSRA	
0058	08	INX	
0059	8C	CPX	
005A	C1	C1	
005B	68	68	
005C	26	BNE	
005D	F7	F7	
005E	3E	WAI	

Figure E12-3  
Program for lighting a display.

## Procedure (Continued)

6. Load the program into the Trainer using the Program Mode and verify that you loaded it correctly.
7. Execute the program using the Program Mode. The left-most digit should display the letter A.
8. The character byte is at address 0051. Change this byte to  $47_{16}$ .
9. Execute the program again. What letter appears in the display?  
\_\_\_\_\_
10. Change the character byte so that the letter H is displayed. What character byte is required? \_\_\_\_\_.
11. Change the character byte to  $79_{16}$ . Execute the program. What character is displayed? \_\_\_\_\_.
12. Refer to Figure E12-4. This figure shows the addresses of the LED's in each of the six displays. You have seen that the left display has an address of  $C16X_{16}$ . The X stands for some number between 0 and F, depending on which segment of that display we wish to use. The next display to the right has an address of  $C15X_{16}$ ; etc. As we mentioned before, bit  $A_3$  is ignored, so  $C16E_{16}$  is the same as  $C166_{16}$ .

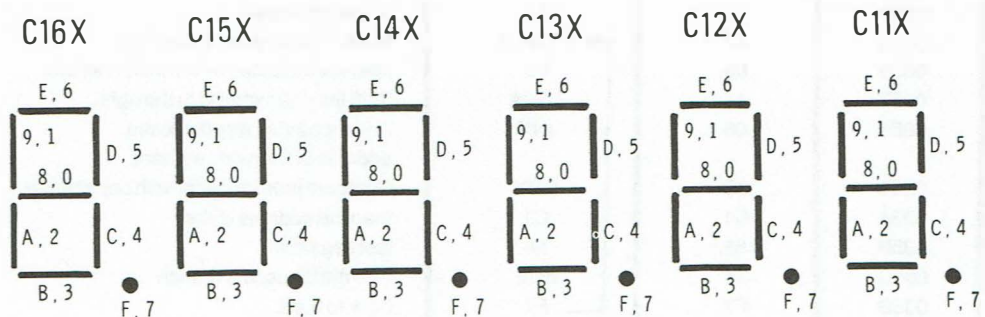


Figure E12-4

Addresses of the various display segments.

13. Now return to the program shown in Figure E12-3. Addresses 0053 and 0054 contain the address of the affected display. By changing this address, we can move the character to a different display. Actually since all display addresses start with C1, we need only change the number at address 0054.
14. Change the byte at 0054 to 50<sub>16</sub>. Change the byte at 005B<sub>16</sub> to 58. Execute the program. The character should appear in the second display from the left.
15. Change the byte at 0054 to 10<sub>16</sub> and the byte at 005B to 18<sub>16</sub>. Execute the program. The character should appear in the right-most display.

## Discussion

It has probably occurred to you that the monitor program must have a subroutine that performs this same function. Fortunately, this subroutine is written in such a way that we can use it. It is called OUTCH for OUTput CHaracter. It starts at address F7C8<sub>16</sub>. You can call this subroutine anytime you like by using the JSR instruction. This subroutine assumes that the character byte is in accumulator A.

## Procedure (Continued)

16. Load the program shown in Figure E12-5. Verify that you loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	86	LDAA#	Load accumulator A immediate with the
0051	37	37	character byte for the letter H.
0052	BD	JSR	Jump to subroutine
0053	F7	F7	OUTCH
0054	C8	C8	
0055	86	LDAA#	Load ACCA with
0056	4F	4F	next character byte.
0057	BD	JSR	
0058	F7	F7	Display it.
0059	C8	C8	
005A	86	LDAA#	Load next character.
005B	0E	0E	
005C	BD	JSR	
005D	F7	F7	Display it.
005E	C8	C8	
005F	86	LDAA#	Load next character.
0060	67	67	
0061	BD	JSR	
0062	F7	F7	Display it.
0063	C8	C8	
0064	3E	WAI	Stop.

Figure E12-5

This program uses the OUTCH subroutine in the monitor program to display a message.

17. Execute the program. What message does the program write?  
\_\_\_\_\_
18. Notice that each character is written in a different display. Thus, the subroutine OUTCH automatically changes the address to that of the next display after each character is written.

## Discussion

The monitor program writes several messages of its own. Examples are: ACCA, ACCB, \_ \_ \_ \_Ad and HEro10. Thus, the monitor has a subroutine that can be used to write messages. It is called OUTSTR for OUTput a STRing of characters. Its starting address is at F7E5<sub>16</sub>. There is a special convention for calling this subroutine. The JSR F7E5<sub>16</sub> instruction must be followed immediately by the character bytes that make up the message. Up to six characters can be displayed. The last character must have the decimal point lit. After the message is displayed, control is returned to the instruction immediately following the last character.

## Procedure (Continued)

19. Load the program shown in Figure E12-6 into the Trainer and verify that you loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	BD	JSR	Jump to the subroutine that will display the following message.  H E L P. ← Decimal point must be lit in last character. Then stop.
0051	F7		
0052	E5		
0053	37	37	
0054	4F	4F	
0055	0E	0E	
0056	E7	E7	
0057	3E	WAI	

Figure E12-6  
The OUTSTR subroutine in the monitor is  
used to display a message.

20. Execute the program. What message does it display? \_\_\_\_\_.
21. Modify the program so that it displays HELLO.



22. The program shown in Figure E12-7 calls the OUTSTR subroutine twice. Load this program into the Trainer.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	BD	JSR	
0051	F7	F7	Call OUTSTR.
0052	E5	E5	
0053	76	76	N
0054	FE	FE	O. ← Decimal point lit (last character).
0055	BD	JSR	
0056	F7	F7	Call OUTSTR again.
0057	E5	E5	
0058	5E	5E	G
0059	FE	FE	O. ← Decimal point lit (last character).
005A	3E	WAI	Then stop.

Figure E12-7  
OUTSTR is called twice.

23. Execute it. What message is displayed? \_\_\_\_\_.
24. Notice that the second message (GO.) is written to the right of the first. Thus, subroutine OUTSTR does not reset the display to the left for the second message.
25. Rewrite the program so that two blank displays appear between NO. and GO.

## Discussion

When long messages such as: "HELLO CAN I HELP YOU?" are being displayed, the display must be given no more than six characters at a time. Also, a short delay must be placed between the various parts of the message. You can achieve a delay by loading the index register with  $FFFF_{16}$  and decrementing it to 0000. You can write a display subroutine and call it between each part of the message.

Also, because we are using the same displays over again for each part of the message, each new word should start on the left. The subroutine called OUTSTR has an alternate entry point at address  $F70A_{16}$  called OUTSTJ. The calling convention for this subroutine is the same as that for OUTSTR. However, each new message starts in the left-most display.

## Procedure (Continued)

26. Load the program shown in Figure E12-8. Verify that you loaded it properly.
27. Execute the program. What message is displayed? \_\_\_\_\_  
\_\_\_\_\_
28. Change the number in address  $0080_{16}$  to  $40_{16}$ .
29. Execute the program. What affect does this have?
30. Write a program of your own that will display "LOAD 2 IS BAD."

## Discussion

The monitor program in the Trainer contains some other useful subroutines. These are outlined in the manual for the ET-18 Robot Trainer. Two of the most useful are REDIS and OUTBYT.

OUTBYT is a subroutine that displays the contents of accumulator A as two hex digits. Its address is  $F7AD_{16}$ . When this subroutine is called for the first time, the two left displays are used. If it is called again without being reset, the two center displays are used. The third time, the two right displays are used.

The display can be reset to the left by calling the REDIS subroutine. This subroutine is located in address  $F64E_{16}$ . If OUTBYT is called after REDIS is called, the two left displays will be used.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	BD	JSR	
0051	F7	F7	Call OUTSTJ.
0052	0A	0A	
0053	37	37	H
0054	4F	4F	E
0055	0E	0E	L
0056	0E	0E	L
0057	FE	FE	O.
0058	BD	JSR	
0059	00	00	Call Delay Subroutine.
005A	7F	7F	
005B	BD	JSR	
005C	F7	F7	Call OUTSTJ again.
005D	0A	0A	
005E	4E	4E	C
005F	77	77	A
0060	76	76	N
0061	00	00	blank
0062	B0	B0	I.
0063	BD	JSR.	
0064	00	00	Call Delay Subroutine.
0065	7F	7F	
0066	BD	JSR	
0067	F7	F7	Call OUTSTJ again.
0068	0A	0A	
0069	37	37	H
006A	4F	4F	E
006B	0E	0E	L
006C	67	67	P
006D	80	80	.
006E	BD	JSR	Call Delay Subroutine.
006F	00	00	
0070	7F	7F	
0071	BD	JSR	Call OUTSTJ again.
0072	F7	F7	
0073	0A	0A	
0074	3B	3B	Y
0075	7E	7E	O
0076	3E	3E	U
0077	00	00	blank
0078	80	80	.
0079	BD	JSR	
007A	00	00	
007B	7F	7F	
007C	7E	JMP	Do it all again.
007D	00	00	
007E	50	50	
007F	CE	LDX#	
0080	FF	FF	} Delay subroutine.
0081	FF	FF	
0082	09	DEX	
0083	26	BNE	
0084	FD	FD	
0085	39	RTS	

Figure E12-8

This program makes extensive use of the subroutine call.

## Procedure (Continued)

31. Load the program shown in Figure E12-9. Verify that you loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	4F	CLRA	Clear accumulator A.
0051	BD	JSR	
0052	F7	F7	Call OUTBYT
0053	AD	AD	
0054	BD	JSR	
0055	00	00	Call Delay Subroutine.
0056	5E	5E	
0057	4C	INCA	Increment accumulator A.
0058	BD	JSR	
0059	F6	F6	Call REDIS
005A	4E	4E	
005B	7E	JMP	
005C	00	00	Do it again.
005D	51	51	
005E	CE	LDX#	
005F	FF	FF	} Delay Subroutine.
0060	FF	FF	
0061	09	DEX	
0062	26	BNE	
0063	FD	FD	
0064	39	RTS	

Figure E12-9  
Using the OUTBYT and REDIS subroutines.

32. Execute the program.
33. Which digits are used by the display? \_\_\_\_\_.
34. Notice that the JSR instruction at address 0058 calls the subroutine that resets the display to the left.
35. To illustrate why this is necessary, let's see what happens when this important step is omitted. Change the contents of locations 0058, 0059, and 005A to 01. This replaces the JSR instruction with three NOPs.



36. Execute the program. Notice that, without calling the REDIS subroutine, the display advances to the right and is lost after the third time through the loop.
  37. Restore the program to its original state. How can the count be speeded up?
- 

## Discussion

You can vary the speed of the count by changing the contents of addresses 005F and 0060. It probably has occurred to you that the Trainer could be turned into a digital clock. In the following procedure, you will develop a program that will do this.

## Procedure

38. Write a program that will count seconds from 00 to 99<sub>10</sub>. The seconds count should be maintained in the two left-most displays. It should count as the above program did, but in decimal instead of hexadecimal.
39. If you have problems, remember that the DAA instruction can be used to convert the addition of BCD numbers to a BCD sum. However, the DAA instruction works only if preceded immediately by an ADDA or ADCA instruction.
40. Load your program into the Trainer and execute it.

## Discussion

One solution is shown in Figure E12-10. Carefully study this program. This routine counts the seconds in decimal. However in a real digital clock, the seconds reset to 00 after 59<sub>10</sub> rather than after 99<sub>10</sub>.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	4F	CLRA	Clear seconds.
0051	BD	JSR	
0052	F7	F7	Call OUTBYT
0053	AD	AD	
0054	BD	JSR	
0055	00	00	Call Delay Subroutine.
0056	60	60	
0057	8B	ADDA#	Increment seconds.
0058	01	01	
0059	19	DAA	Make it decimal.
005A	BD	JSR	
005B	F6	F6	Call REDIS
005C	4E	4E	
005D	7E	JMP	
005E	00	00	Do it all again.
005F	51	51	
0060	36	PSHA	
0061	86	LDAA#	
0062	02	02	
0063	CE	LDX#	
0064	B0	B0	
0065	A3	A3	
0066	09	DEX	One-Second
0067	26	BNE	
0068	FD	FD	Delay Subroutine
0069	4A	DECA	
006A	26	BNE	
006B	F7	F7	
006C	32	PULA	
006D	39	RTS	

Figure E12-10

This routine counts seconds from 00 to 99.

## Procedure (Continued)

41. Modify your program (or the one in this experiment) so that it displays seconds from 00 to 59 and then returns to 00 and starts over again.
42. Load your program into the Trainer and execute it.
43. Debug your program if necessary until it performs properly.

STANDARD ADDRESS	STANDARD ADDRESS	STANDARD ADDRESS	STANDARD ADDRESS
0000	0000	0000	0000
0001	0001	0001	0001
0002	0002	0002	0002
0003	0003	0003	0003
0004	0004	0004	0004
0005	0005	0005	0005
0006	0006	0006	0006
0007	0007	0007	0007
0008	0008	0008	0008
0009	0009	0009	0009
0010	0010	0010	0010
0011	0011	0011	0011
0012	0012	0012	0012
0013	0013	0013	0013
0014	0014	0014	0014
0015	0015	0015	0015
0016	0016	0016	0016
0017	0017	0017	0017
0018	0018	0018	0018
0019	0019	0019	0019
0020	0020	0020	0020
0021	0021	0021	0021
0022	0022	0022	0022
0023	0023	0023	0023
0024	0024	0024	0024
0025	0025	0025	0025
0026	0026	0026	0026
0027	0027	0027	0027
0028	0028	0028	0028
0029	0029	0029	0029
0030	0030	0030	0030
0031	0031	0031	0031
0032	0032	0032	0032
0033	0033	0033	0033
0034	0034	0034	0034
0035	0035	0035	0035
0036	0036	0036	0036
0037	0037	0037	0037
0038	0038	0038	0038
0039	0039	0039	0039
0040	0040	0040	0040
0041	0041	0041	0041
0042	0042	0042	0042
0043	0043	0043	0043
0044	0044	0044	0044
0045	0045	0045	0045
0046	0046	0046	0046
0047	0047	0047	0047
0048	0048	0048	0048
0049	0049	0049	0049
0050	0050	0050	0050
0051	0051	0051	0051
0052	0052	0052	0052
0053	0053	0053	0053
0054	0054	0054	0054
0055	0055	0055	0055
0056	0056	0056	0056
0057	0057	0057	0057
0058	0058	0058	0058
0059	0059	0059	0059
0060	0060	0060	0060
0061	0061	0061	0061
0062	0062	0062	0062
0063	0063	0063	0063
0064	0064	0064	0064
0065	0065	0065	0065
0066	0066	0066	0066
0067	0067	0067	0067
0068	0068	0068	0068
0069	0069	0069	0069
0070	0070	0070	0070
0071	0071	0071	0071
0072	0072	0072	0072
0073	0073	0073	0073
0074	0074	0074	0074
0075	0075	0075	0075
0076	0076	0076	0076
0077	0077	0077	0077
0078	0078	0078	0078
0079	0079	0079	0079
0080	0080	0080	0080
0081	0081	0081	0081
0082	0082	0082	0082
0083	0083	0083	0083
0084	0084	0084	0084
0085	0085	0085	0085
0086	0086	0086	0086
0087	0087	0087	0087
0088	0088	0088	0088
0089	0089	0089	0089
0090	0090	0090	0090
0091	0091	0091	0091
0092	0092	0092	0092
0093	0093	0093	0093
0094	0094	0094	0094
0095	0095	0095	0095
0096	0096	0096	0096
0097	0097	0097	0097
0098	0098	0098	0098
0099	0099	0099	0099

## Discussion

One solution is shown in Figure E12-11. The seconds count is compared to 60 each time it is incremented. When it reaches 60, it is reset to 00.

The next step is to add a minutes count. You can do this by incrementing a decimal number each time the seconds count “rolls over” from 59 to 00. The decimal number is then displayed as minutes.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	C6	LDAB#	Load number for comparison.
0051	60	60	
0052	4F	CLRA	Clear seconds.
0053	BD	JSR	
0054	F7	F7	Call OUTBYT
0055	AD	AD	
0056	BD	JSR	
0057	00	00	Call Delay Subroutine.
0058	64	64	
0059	BD	JSR	
005A	F6	F6	Call REDIS
005B	4E	4E	
005C	8B	ADDA#	Increment seconds.
005D	01	01	
005E	19	DAA	Make it decimal.
005F	11	CBA	Time to clear seconds.
0060	27	BEQ	Yes.
0061	F0	F0	
0062	20	BRA	No.
0063	EF	EF	
0064	36	PSHA	
0065	86	LDAA#	
0066	02	02	
0067	CE	LDX#	
0068	B0	B0	
0069	A3	A3	
006A	09	DEX	
006B	26	BNE	
006C	FD	FD	
006D	4A	DECA	
006E	26	BNE	
006F	F7	F7	
0070	32	PULA	
0071	39	RTS	

Figure E12-11

This routine counts seconds from 00 to 59.

## Procedure (Continued)

44. Write a program that will display minutes and seconds properly. The minutes should be displayed in the two left displays; the seconds in the two center displays. Like the seconds, the minutes should return to 00 after 59.
45. Load your program and execute it.
46. Debug your program as necessary.



## Discussion

A solution is shown in Figure E12-12. Your approach may be more straightforward, but may require more memory. Notice that the first two addresses are data. The program starts at address 0052.

The final step is to include the hours display.

## Procedure (Continued)

47. Modify your program so that it displays hours, minutes and seconds.
48. Load your program and execute it.
49. Debug your program as necessary.

A solution is shown in Figure E12-13. This program evolved over a period of time and is extremely compact. It is virtually impossible for a beginning programmer to write a program this compact on the first try. Your program may require substantially more memory, but the important thing is: does it work? This time the first three addresses are data. The program starts at 0053.

You can “fine tune” the clock period by changing the numbers at 0056<sub>16</sub> and 0057<sub>16</sub>. Although this is a crystal-controlled clock, it will require some effort to make it accurate. The ET-18 Robot Trainer has a highly accurate calendar clock. In another experiment you’ll learn how to access it.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	00	00	Reserved for seconds.
0051	00	00	Reserved for minutes.
0052	36	→ PSHA	} One-Second Delay
0053	86	LDAA#	
0054	02	02	
0055	CE	LDX#	
0056	B0	B0	
0057	A3	A3	
0058	09	DEX	
0059	26	BNE	
005A	FD	FD	
005B	4A	DECA	
005C	26	BNE	} Load number for comparison.
005D	F7	F7	
005E	32	PULA	
005F	C6	LDAB#	
0060	60	60	
0061	0D	SEC	Set carry bit.
0062	8D	BSR	Branch to subroutine to
0063	11	11	increment seconds.
0064	8D	BSR	Branch to the same subroutine
0065	0F	0F	to increment minutes.
0066	BD	JSR	
0067	F6	F6	Call IREDIS
0068	4E	4E	
0069	96	LDAA	Load minutes.
006A	51	51	
006B	BD	JSR	
006C	F7	F7	Call OUTBYT to
006D	AD	AD	display minutes.
006E	96	LDAA	Load seconds
006F	50	50	
0070	BD	JSR	Call OUTBYT TO
0071	F7	F7	display seconds
0072	AD	AD	
0073	20	BRA	Do it all again.
0074	DD	DD	
0075	A6	LDAA, X	Load seconds (or minutes) into A.
0076	50	50	
0077	89	ADCA#	Increment if necessary
0078	00	00	
0079	19	DAA	Adjust to decimal
007A	11	CBA	Time to clear?
007B	26	BNE	No.
007C	01	01	
007D	4F	CLRA	Yes.
007E	A7	STAA, X	Store seconds (or minutes)
007F	50	50	
0080	08	INX	
0081	07	TPA	
0082	88	EORA#	Complement carry bit
0083	01	01	
0084	06	TAP	
0085	39	RTS	

Increment subroutine

Figure E12-12

Routine for displaying minutes and seconds.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	00	00	Reserved for seconds.
0051	00	00	Reserved for minutes.
0052	00	00	Reserved for hours.
0053	36	PSHA	One-Second Delay Subroutine.
0054	86	LDAA#	
0055	02	02	
0056	CE	LDX#	
0057	B0	B0	
0058	A3	A3	
0059	09	DEX	
005A	26	BNE	
005B	FD	FD	
005C	4A	DECA	
005D	26	BNE	Minutes and seconds will be compared with sixty. Prepare to increment seconds. Go to subroutine that will increment seconds. Go to same subroutine. It will increment minutes if necessary. Hours will be compared with twelve. Go to same subroutine. It will increment hours if necessary.
005E	F7	F7	
005F	32	PULA	
0060	C6	LDAB#	
0061	60	60	
0062	0D	SEC	
0063	8D	BSR	
0064	11	11	
0065	8D	BSR	
0066	0F	0F	
0067	C6	LDAB#	Call REDIS
0068	12	12	
0069	8D	BSR	
006A	0B	0B	
006B	BD	JSR	
006C	F6	F6	
006D	4E	4E	
006E	8D	BSR	
006F	17	17	
0070	8D	BSR	
0071	15	15	Call display subroutine to display hours. Call display subroutine to display minutes. Call display subroutine to display seconds. Do it all again.
0072	8D	BSR	
0073	13	13	
0074	20	BRA	
0075	DD	DD	
0076	A6	LDAA, X	
0077	50	50	
0078	89	ADCA#	
0079	00	00	
007A	19	DAA	
007B	11	CBA	Adjust to decimal. Time to clear? No. Yes. Store seconds (or minutes or hours). Point index register at minutes (or hours). Complement carry bit.
007C	25	BCS	
007D	01	01	
007E	4F	CLRA	
007F	A7	STAA, X	
0080	50	50	
0081	08	INX	
0082	07	TPA	
0083	88	EORA#	
0084	01	01	
0085	06	TAP	Point index register at hours (or minutes or seconds). Load hours (or minutes or seconds). Display hours (or minutes or seconds).
0086	39	RTS	
0087	09	DEX	
0088	A6	LDAA, X	
0089	50	50	
008A	7E	JSR	
008B	F7	F7	
008C	AD	AD	
008D	39	RTS	

Figure E12-13  
Twelve-hour clock program.

# Experiment 13

## *Sensors*

**OBJECTIVES:**

*To observe the operation of a light sensor.*

*To vary the sensitivity of the light sensor to compensate for a change in ambient light.*

*To measure range and determine the accuracy of your ultrasonic sensor.*

*To determine the minimum range of your ultrasonic sensor.*

## **Introduction**

In this unit of the course you learned how sensors are used as the eyes and ears of a robot. You also learned how temperature and ultrasonic sensing is used in the industrial setting. In this experiment you will become familiar with the operation of some of these sensors.

In the first portion of this experiment, you will see how you can adjust the light sensor's sensitivity to compensate for ambient room light.

In the last portion of this experiment, you will observe how you can use an ultrasonic sensor to measure the distance to an object.

This experiment is meant to show the basic operation of some of the sensing capabilities of the Trainer. In Experiment 17, you will be shown in much greater detail how these sensors are actually used to control a robotic function.

## **Material Required**

ET-18 Robot Trainer  
Flashlight (not provided)  
Yardstick (not provided)



## Procedure

1. Observing Figure E13-1, carefully remove the back panel from the Trainer. Note the position of the sense board. Also note the position of the LIGHT Sensitivity adjustment and the eight LEDs.
2. Position the LIGHT Sensitivity adjustment to its center position.
3. Energize and initialize the Robot Trainer. Note that the eight LEDs are off.

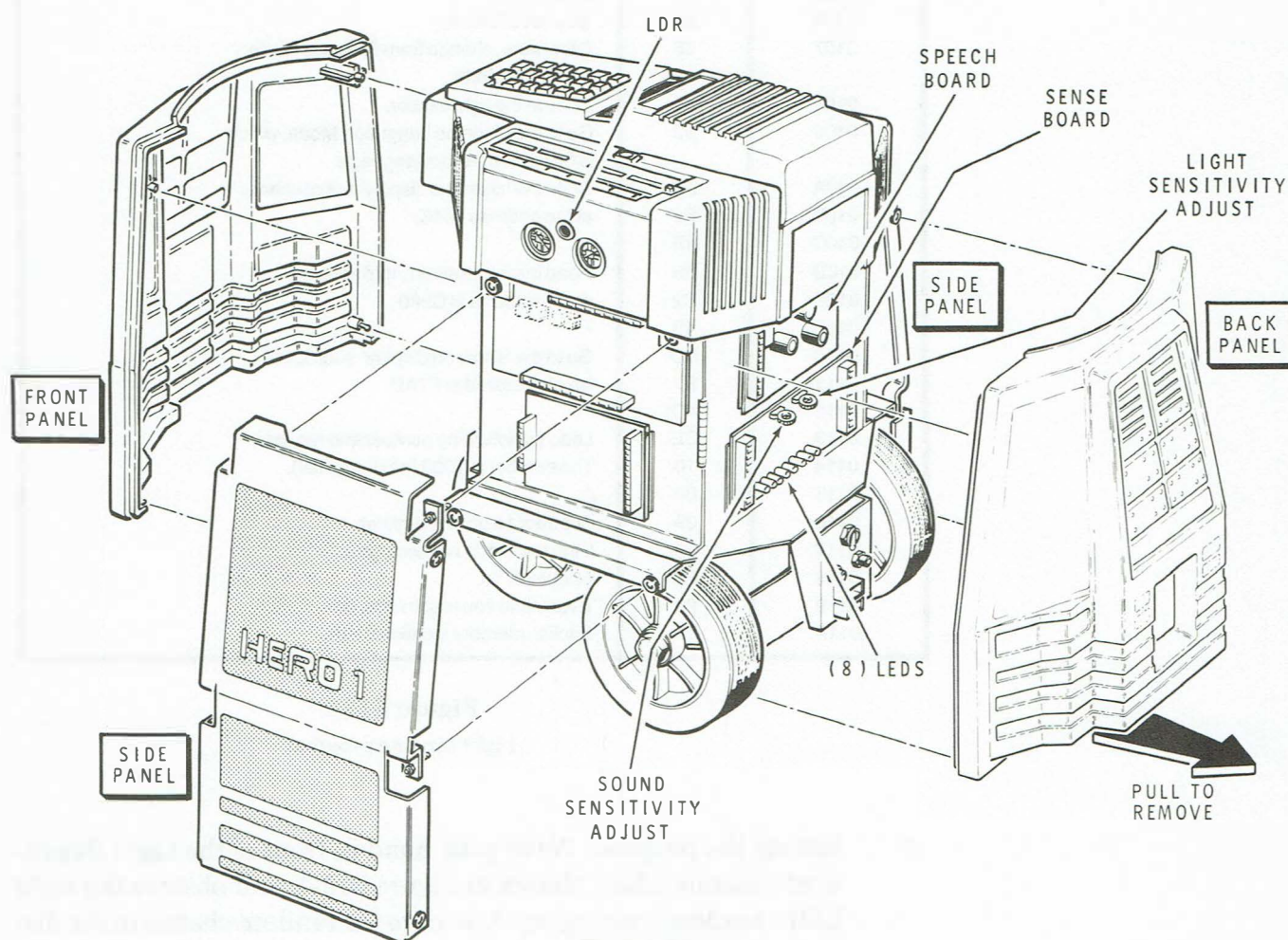


Figure E13-1

4. Using the keyboard, enter the program shown in Figure E13-2. Once you have entered the program, press the **RESET** key. Go back and examine the program for errors.

MEMORY ADDRESS	INPUT	COMMENTS
0100	B6	Read the number...
0101	0E	which is found at 0EE1.
0102	E1	(0 = REPEAT MODE).
0103	81	Compare that number...
0104	00	to zero.
0105	26	If not equal...
0106	01	go ahead to 0108.
0107	3F	Otherwise, change from Program Mode to Repeat Mode.
0108	41	Turn on the light sensor.
0109	83	Go to the Machine Language Mode, which is faster than Robot language.
010A	BD	Go to the "clear the display" subroutine... at the address F64E.
010B	F6	
010C	4E	/
010D	B6	Read the information (light sensor)...
010E	C2	which is found at C240.
010F	40	/
0110	BD	Go to the "show on display" subroutine...
0111	F7	which is stored at F7AD.
0112	AD	/
0113	CE	Load the following number into register.
0114	10	The number is 1000 (hexadecimal).
0115	00	/
0116	09	Subtract 1 from the register.
0117	26	If the register is not zero, go...
0118	FD	back to 0116.
0119	20	Every time you reach here, go...
011A	EF	back to memory location 010A.

Figure E13-2  
Light sensing program.

5. Initiate the program. Wave your hand in front of the Light Dependent Resistor (LDR), shown in Figure E13-1, and observe the eight LED's randomly changing. Also note the random change in the display.

6. Shine your flashlight directly at the LDR so that all eight LED's are on. Note and record the reading shown on the display.

Display Reading \_\_\_\_\_

7. Using your hand, block all light going to the LDR until all eight LED's are off. (You may have to dim the ambient room light to obtain this reading). Note and record the reading shown on the display.

Display Reading \_\_\_\_\_

8. Leaving the LDR exposed to ambient room light, record the reading shown on the display.

Display Reading \_\_\_\_\_

9. Now turn the LIGHT sensitivity adjustment all the way to its RIGHT stop and record the reading shown on the display.

Display Reading \_\_\_\_\_

Did the display reading increase or decrease in value?

\_\_\_\_\_

10. Adjust the LIGHT Sensitivity to the left until you have the approximate value shown on the display that you recorded in Step 8.
11. Position the LIGHT Sensitivity adjustment all the way to its LEFT stop and record the reading shown on the display.

Display Reading \_\_\_\_\_

Did the display reading increase or decrease in value, from the value shown in Step 8.

\_\_\_\_\_

12. Position the LIGHT Sensitivity adjustment back to its center position, reset the program, and then read the following discussion.

## Discussion

Figure E13-3 is a simplified block diagram of the light sensing circuit used in the Robot Trainer. R1, R2, and the LDR form a voltage divider network, which sets the bias level for the amplifier. Recall that the resistance of an LDR changes with a change in the light level applied to it. Therefore, the analog signal from the LDR, representing a specific level of light, is sent from the amplifier to the A-to-D converter where it is changed into a digital, binary equivalent. The output of the A-to-D converter is sent to the eight LEDs and the MPU.

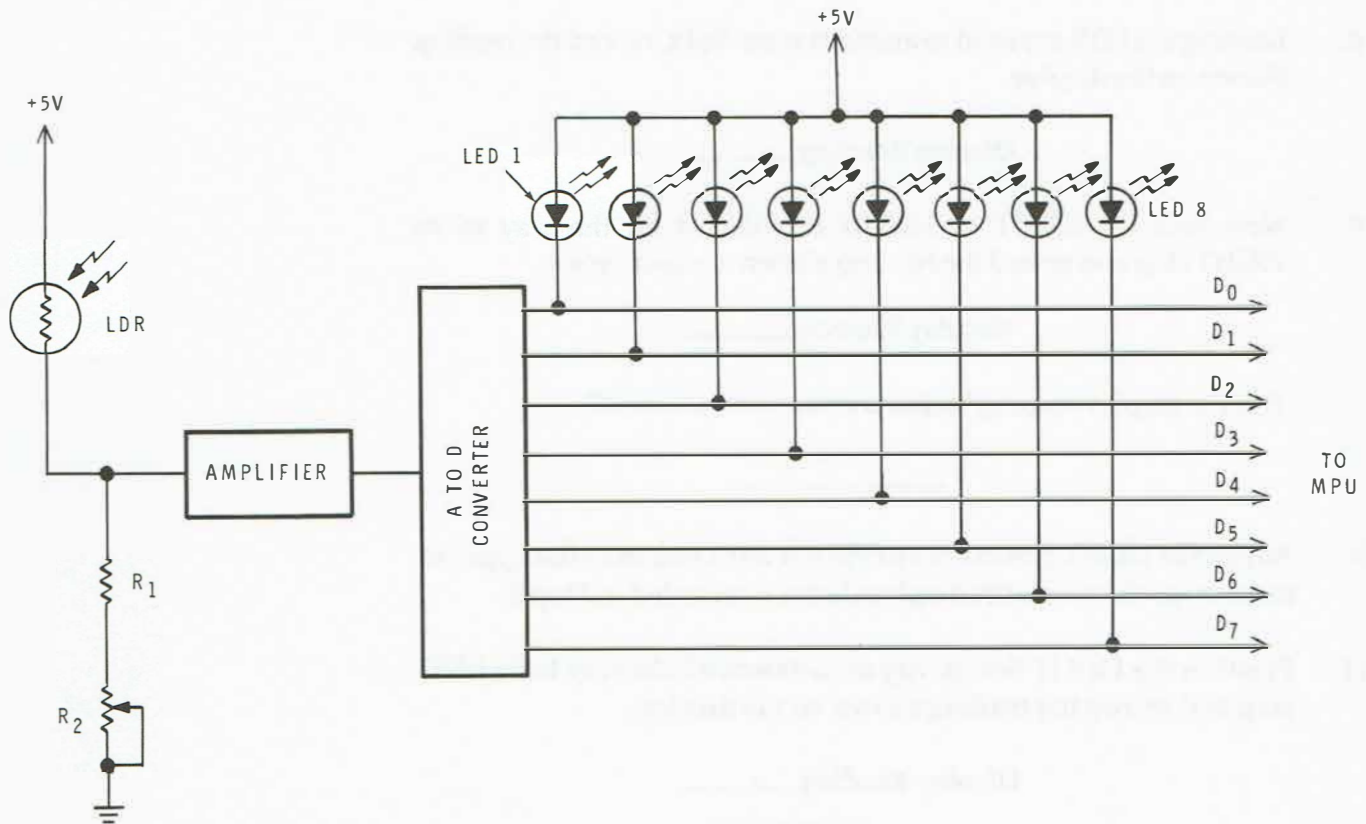


Figure E13-3  
Block diagram of light sensing circuit.

When you entered the program, you basically told the MPU to continuously check the output of the light sensing circuit and send the data to the display. Remember, the MPU takes binary information and converts it to hexadecimal. Thus, data shown on the display was the hexadecimal equivalent to the value shown on the eight LEDs. In Step 6 of the procedure, when all eight LEDs were on, representing the decimal value 256, the hexadecimal value shown on the display was FF, the equivalent to decimal value 256.

As you noted in procedure Steps 6, 7, and 8, when the LIGHT Sensitivity level was set to its midrange position, the LDR could sense 256 different light levels ranging from 0 to 256. However, in later steps when you positioned the LIGHT Sensitivity level to its upper and lower stops, the biasing reference to the amplifier was changed and the LDR did not sense the same 256 levels of light.

The Robot Trainer can be programmed to perform a specific function when the light sensor detects a specific level of light. However, as you have seen, the ambient light level of the room could affect this function. Thus, you may desire to adjust the LIGHT Sensitivity level to compensate for this, instead of changing the MPU program.



## Procedure (Continued)

13. Enter and examine the program shown in Figure E13-4.

MEMORY ADDRESS	INPUT	COMMENTS
0200	B6	Read the number...
0201	0E	which is found at 0EE1.
0202	E1	(0 = Repeat Mode).
0203	81	Compare that number...
0204	00	to zero.
0205	26	If not equal...
0206	01	go ahead to 0208.
0207	3F	Otherwise, change from Program Mode to Repeat Mode.
0208	45	Turn on, and begin sampling sonar. Store reading at special place (0010) in memory.
0209	83	Go to Machine Language Mode, which is much faster than operating in the Robot language.
020A	BD	Jump to "clear display" subroutine...
020B	F6	at the address F64E.
020C	4E	/
020D	96	Load into computer the contents of...
020E	11	special address (0011) (sonar reading).
020F	BD	Jump to "print on display" subroutine...
0210	F7	(which is at F7AD).
0211	AD	/
0212	CE	Load the following number in register.
0213	10	The number is 1000 (hexadecimal).
0214	00	/
0215	09	Subtract one from the number.
0216	26	If the number is not yet zero go...
0217	FD	back to 0215.
0218	20	Every time you reach here, go...
0219	F0	back to memory 020A.

Figure E13-4  
Ultrasonic ranging program.

14. Place the Robot Trainer in such a manner that is pointing its ultrasonic sensor, located in the front of the turret, directly at a solid object approximately 16" away. (For best results, point the Trainer at a wall.)
15. Execute the program. You will see some number shown on the display. Using the display as your guide, carefully move the Trainer forwards or backwards until you obtain a reading of 30 on the display.
16. Using the yardstick, measure and record the distance in inches from the front of the turret to the wall. (Be sure the display remains at 30 while you are taking this measurement.)

Distance \_\_\_\_\_

17. Again using the display as your guide, move the Trainer backwards until you have a reading of 40 on the display.
18. Measure and record the distance from the front of the turret to the wall.

Distance \_\_\_\_\_

19. Subtract the reading you obtained in Step 16 from the reading you obtained in Step 18, and record the difference.

Difference \_\_\_\_\_

20. Divide the difference from Step 19 by 10 and record the quotient.

Quotient \_\_\_\_\_

21. While observing the display, slowly move the Trainer towards the wall until the reading starts to become erratic. Record the last valid reading before the readings became erratic. (You may wish to try this several times to obtain a valid reading.)

Reading \_\_\_\_\_

22. Reset and secure the Trainer. Then read the following discussion.

## Discussion

When you entered the program, you basically told the MPU to continuously check the output of the ultrasonic sensing circuit and send the data to the display. In the first part of the procedure, you determined the ultrasonic ranging capabilities of your Trainer by moving it farther away from the object. Recall that all values shown on the display are in hexadecimal. Therefore, when you moved the Trainer 10 hexadecimal values farther away from the object and divided the difference of the two readings by 10, you determined the value in inches of each hexadecimal value for your Trainer.

This information would be valuable if you desired to program your Trainer so that it would come no closer than a specified distance to an object. However, as you were shown in the last part of the experiment, the minimum distance the Trainer can distinguish is also important. Thus, the Trainer could only come so close to an object before the range information would become invalid. It should be mentioned before we leave this discussion that the maximum range the Trainer can measure is approximately 9 feet.

## Experiment 14

### *“Open-Loop” Control of a DC Stepping Motor*

**OBJECTIVES:**

*To illustrate a method of manually single-stepping a DC stepper motor in the "open-loop" control mode.*

## Introduction

In Unit Eight you learned how both open-loop (feedback signals not required) and closed-loop (feedback signals required) servomechanisms are used for control purposes. During our discussion of these two types of control systems, it was generally accepted that the closed-loop system provides a greater degree of control. However, there is one type of open-loop control system used in the Robot Trainer, the DC stepper motor, that provides very precise control over a given function.

As you recall from your previous studies, stepper motors are becoming increasingly popular in industrial applications, especially robotics, where computer control is the norm. They offer significant advantages over the usual closed-loop servomotor systems found in many of the older industrial settings. With open-loop steppers, feedback signals are not required, and motor error is noncumulative as long as pulse-to-step integrity is maintained.

In this experiment, you will observe how and why open-loop stepper motors are an excellent choice for control applications.

## Materials Required

ET-18 Robot Trainer

Logic Probe (connected on experimental board in Experiment 3)

4 3300 ohm, 5 percent, 1/4-watt resistors

1 8-section, SPST, DIP switch

Hookup wire (22 gauge, white, solid conductor)



## Procedure

1. Construct the “open-loop” manual stepping control circuit shown in Figure E14-1 on the experimental board. Use DIP switches 1, 3, 5, and 7. NOTE: resistors R1, 2, 3, and 4 are connected to data input lines DI<sub>0</sub> through DI<sub>3</sub> respectively. Data input lines DI<sub>4</sub> through DI<sub>7</sub> are tied to ground. Ensure that all switches are in the “ON” position.

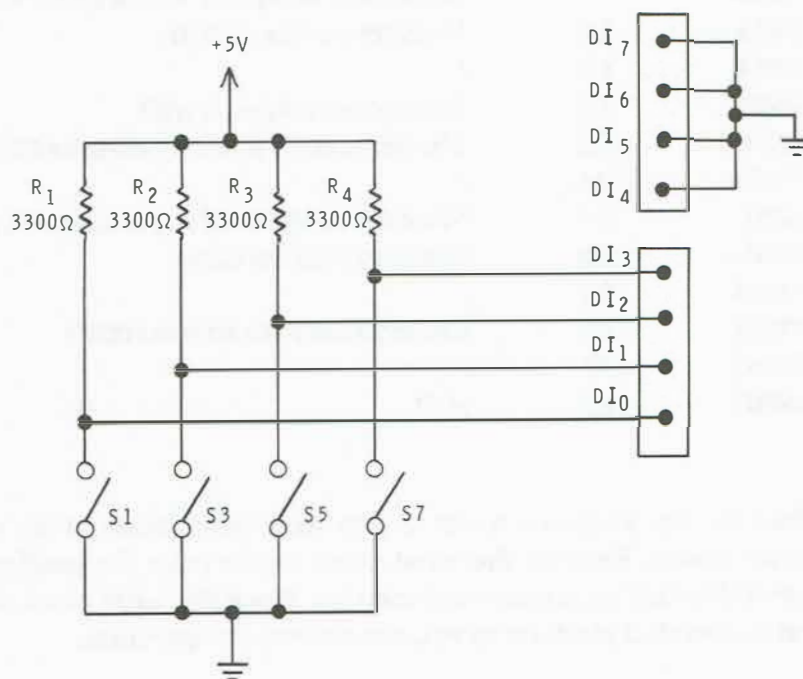


Figure 14-1  
Circuit diagram for “open-loop” manual  
stepping control.

2. Energize and initialize the Robot Trainer.

3. Enter the program shown below, using the Direct Programming Mode. Remember to examine the program for errors. DO NOT EXECUTE THE PROGRAM UNTIL INSTRUCTED TO DO SO.

MEMORY ADDRESS	INPUT	COMMENT
0050	C6	Load accumulator B with
0051	40	The number 40 (hexadecimal)
0052	F7	Store the contents of accumulator B at
0053	C2	Memory address C2C0
0054	C0	/
0055	B6	Load accumulator A with
0056	C2	The contents at memory address C2A0
0057	A0	/
0058	B7	Store the contents of accumulator A at
0059	C2	Memory address C260
005A	60	/
005B	20	Branch always to address 0050
005C	F3	/
005D	E3	Halt

4. Position the Trainer's turret so you have easy access to the wrist pivot motor. Remove the wrist pivot motor from the gearbox by carefully turning it counterclockwise. Place the wrist pivot motor on an elevated platform so you can observe its operation.
5. Using a 2 1/2" piece of transparent tape, construct an indicating flag on the geared shaft of the wrist pivot motor, as shown in Figure E14-2. This flag will enable you to observe the stepping action of the motor.

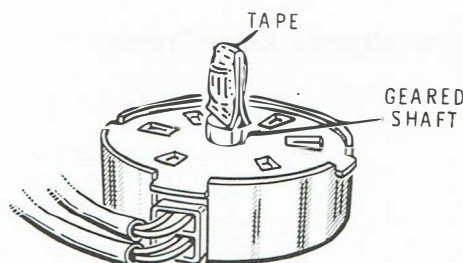


Figure E14-2  
Position indicating flag.

6. Execute the program you entered in Step 3.
7. Locate P701 on the Arm Drive circuit board and, using the logic probe, check the wrist pivot motor "enable" signal at pin 19. Is the logic level high or low? \_\_\_\_\_
8. In the same manner, check and record the logic level of the motor drive signals at pins 9, 10, 11, and 12, at P701.

Pin 9 \_\_\_\_\_

Pin 10 \_\_\_\_\_

Pin 11 \_\_\_\_\_

Pin 12 \_\_\_\_\_

9. On the experimental board, place switches 1, 3, 5, and 7 in the off position. Check and record the logic levels at pins 9, 10, 11, and 12 of P701.

Pin 9 \_\_\_\_\_

Pin 10 \_\_\_\_\_

Pin 11 \_\_\_\_\_

Pin 12 \_\_\_\_\_

10. Return switches 1, 3, 5, and 7 to the on position. Using the sequence chart shown in Figure E14-3 and observing the motor, position the switches as shown in Step 1. Did the motor move?  
\_\_\_\_\_
11. Continue stepping the motor by positioning the switches as shown in Step 2 of Figure E14-3. Then Step 3, and finally Step 4. Does the motor rotate clockwise or counterclockwise?  
\_\_\_\_\_

CW ROTATION ↓	STEP	SW 1	SW 3	SW 5	SW 7	CCW ROTATION ↑
	1	ON	OFF	ON	OFF	
	2	ON	OFF	OFF	ON	
	3	OFF	ON	OFF	ON	
	4	OFF	ON	ON	OFF	
	1	ON	OFF	ON	OFF	

Figure E14-3

Manual stepping sequence chart.

12. Place all switches in the on position.
13. Still referring to Figure E14-3, step the motor by placing the switches to the position shown in Step 4, followed by Steps 3, 2, and 1. Did the motor step in the opposite direction? \_\_\_\_\_
14. Turn off power to the Trainer, replace the wrist pivot motor, and remove the manual stepping circuit from the experimental board. Then read the following discussion.

## Discussion

In this experiment you manually controlled one of the Trainer's stepping motors. All stepping motors in the Robot Trainer are controlled in the "open-loop" mode. That is, only the pulses control the motor, and no feedback or error signals are used. You observed how the sequence in which the pulses were sent to the motor controlled the motor's direction of rotation. In the same manner, the amount of rotation is determined by how long you keep sequencing the motor.

This open-loop method of precise control is possible since the motor will only step one increment for each set of pulses applied. The motor used in this experiment is a 15 degree stepper; therefore, each step of pulses will rotate the motor exactly 15 degrees. Since the motor will only step 15 degrees at a time, the exact position of the motor is known at all times. These motors can also be connected to a gearbox to reduce the amount of travel even further, as is the case in the Robot Trainer. The gearbox not only permits more precise control; it also permits a small motor to drive a larger load by amplifying its torque.

Stepper motors used in the open-loop mode lend themselves well to robotic applications where a microprocessor is used as the controller. The microprocessor can easily control the sequence and the duration of the pulses applied, thus determining the direction and amount of motor rotation. Also, the speed of rotation is easily controlled by simply varying the speed at which the pulses are sent to the motor. As you can see the direction, amount, and speed of rotation, are all easily controlled without the use of feedback circuitry.

# Experiment 15

## *Robot Voice Routine*



**OBJECTIVES:**

*To write programs that invoke the Robot's voice.*

*To use phrases from the Robot's ROM.*

*To construct phoneme trees for the Robot.*

*To use inflection on phonemes.*

*To identify the hardware that controls the Robot's voice.*

## Introduction

The Robot Trainer's voice is provided by a phoneme speech synthesis chip. You've heard the Robot's voice say "Ready" every time you press **RESET**. In this experiment you will learn how you can make it say what you want. You'll learn how to access the phoneme phrases in ROM, and how to build phoneme phrases of your own. You'll learn how to access a word or phrase several times to conserve memory. You will work with three of the aspects of voice synthesis (that you learned in Unit 9) in this experiment: inflection, timing, and phoneme selection.

Inflection deals with the relative pitch, or frequency, and volume of one phoneme to the others in its word or phrase. You cannot dynamically control the volume of the Robot's phonemes, but you can control the pitch. The speech synthesizer allows you to select any one of four pitch values for each phoneme in the string. By using different values, you can make the voice more or less expressive.

Timing is another significant factor in voice production. You can draw out a phoneme to emphasize it. You can also affect meaning by using silence, putting pauses in important places.

Finally, you change meaning by the phonemes you use in a word. Actually, you may be changing the word itself. For example, the word "read", means one thing if the "ea" is pronounced as in "reed", but something else if it's pronounced as in the word "red." Also, people in different parts of the country pronounce words different ways. By changing the phonemes, you can try to give the Robot an accent.

In this experiment, you will also see how the Robot works with phoneme strings. In addition to the phonemes you read about in Unit 9, the Robot has two special phonemes. By using these special phonemes, the Trainer's voice program can read phoneme strings directly, or branch to phrases through a "Speech Tree." The special phonemes are actually different inflections applied to the STOP phoneme ( $3F_{16}$ ). With the high bit set ( $BF_{16}$ ), the phoneme is interpreted as a subroutine branch to the phoneme string at the address indicated by the next two bytes. A maximum of eight levels are permitted for phoneme subroutine strings. With both high bits set ( $FF_{16}$ ), the phoneme string is ended. If this was the end of a subroutine called by  $BF_{16}$ , the  $FF_{16}$  is taken as a **return** to the next phoneme after the branch. If the string is not a subroutine, the speech is ended. So, every phrase should use this end phoneme. Further, every word should be followed by a "no sound" phoneme. Therefore, you will normally precede every  $FF_{16}$  with a  $3F_{16}$  or  $3E_{16}$  phoneme.

The ET-18 Robot's Repeat Mode allows you to work with phoneme speech very easily. You will use this mode here to work with speech trees. Later, you will learn variations of the Repeat Mode speech format that allow you to point to the string with the index register, or do other things while the Robot is speaking. For now, you will use the code that speaks the phrase at an indicated location in memory. The code for this operation is  $72\ xxxx$ , where  $xxxx$  is the starting address of the phoneme speech tree.

## Material Required

ET-18 Robot Trainer with Voice Module

## Procedure

For all programs in this experiment, you must use the Repeat Mode of operation.

1. Enter the program listed in Figure E15-1.
2. Set the program counter to 0050 and single step through the program. Notice that every time the 72 F4A1 instruction is executed the robot says "READY."

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	72	SPKwait (READY)	Say READY
0051	F4		
0052	A1		
0053	20	BRA	Do it again
0054	FB		

Figure E15-1

## Discussion

The SPKwait instruction, 72 xxxx, causes the robot to speak the phoneme string starting at address xxxx. F4A1 is the address for "READY." Thus, each time 72 F4A1 is executed, the Robot says "READY." If you DO 0050, the Robot will continue to say "READY" until you press **RESET**. You can find the starting address for the "READY" phrase and several others in the Robot Voice Dictionary that came with your Robot Voice package.

## Procedure (Continued)

3. Enter the program in Figure E15-2.
4. Set the program counter to 0050 and single step through this program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	72	SPKwait	Say the phrase
0051	01	(phrase)	
0052	00		
0053	20	BRA	
0054	FB		
			Do it again
0100	BF	Branch	
0101	F4	(READY)	
0102	A1		
0103	BF	Branch	
0104	FA	(I CAN TALK, LIKE THIS)	Phoneme tree branch
0105	93		
0106	FF	END	

Figure E15-2

## Discussion

In this program, the phoneme list starts at address 0100. We've used the special phoneme, BF, to branch to the "READY" phrase, and the "I CAN TALK, LIKE THIS" phrase. Notice that we've used the FF phoneme to end the phoneme phrase. Now you are familiar with the procedure for speaking the "canned phrases" in the Robot's ROM. If you've looked at the Robot Voice Dictionary, you may have noticed that it also contains the phoneme listing for many words. Here's how you can use these.

## Procedure (Continued)

5. Enter the program listed in Figure E15-3. Notice that the main program is the same as Figure E15-2.
6. Single step this program. What did the robot say?

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	72	SPKwait	Say the phrase
0051	01	(phrase)	
0052	00		
0053	20	BRA	Do it again
0054	FB		
0100	BF	Branch	Phoneme tree branch
0101	01	(phrase)	
0102	03		No sound
0103	3E	/PA1/	
0104	DB	/H/	
0105	42	/EH1/	
0106	63	/UH3/	
0107	58	/L/	
0108	63	/UH3/	
0109	74	/O2/	
010A	B7	/U1/	
010B	3E	/PA1/	
010C	FF	END	

Figure E15-3

## Discussion

This time the phoneme string is entered directly. It is spoken twice for each single step, since it is first called as a routine and then the phoneme tree “falls into” the phoneme string to say it again. You may have recognized this string from Unit 9. It says “HELLO.” You should now understand how to enter phoneme strings, access phrases in ROM, and use the branch phoneme to construct a phoneme tree. Now let’s experiment with the inflection codes and the controls on the Robot’s voice synthesizer board.



## Procedure (Continued)

7. Enter the program listed in Figure E15-4.
8. Execute the program.
9. Press **RESET** to stop the program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	72	SPKwait	Say the phrase
0051	01	(phrase)	
0052	00		
0053	86	LDAA	Load value into A
0054	40	40	0100 0000 Binary
0055	BB	ADDA	Add A to memory
0056	01	01	address
0057	0B	0B	010B
0058	B7	STAA	Store A at
0059	01	01	address
005A	0B	0B	010B
005B	20	BRA	Do it again
005C	F3		
0100	BF	Branch	Phoneme tree branch
0101	01	(phrase)	
0102	0B		
0103	BF	Branch	Phoneme tree branch
0104	01	(phrase)	
0105	0B		
0106	BF	Branch	Phoneme tree branch
0107	01	(phrase)	
0108	0B		
0109	3E	/PA1/	No sound
010A	FF	END	
010B	24	/AH/	Say Ah!
010C	FF	END	

Figure E15-4

## Discussion

This program demonstrates the inflection controls to the phoneme speech synthesizer. You should hear four distinct tones, starting with a low tone and working to a high one. These correspond to the inflection values applied to the phoneme AH, 24<sub>16</sub>. The instruction at address 0053 sets the A<sub>6</sub> bit in the A register. This is added to the inflection (0055) and stored as the new phoneme (0058). Each time through the loop, the inflection bits, A<sub>6</sub> and A<sub>7</sub>, are incremented as follows:

<u>Pass</u>	<u>Inflection</u>				<u>Phoneme</u>
First	00	10	0100	Binary	
Second	01	10	0100	"	
Third	10	10	0100	"	
Fourth	11	10	0100	"	
Fifth	00	10	0100	"	
(Etcetera)					

Now try something similar with a word.

## Procedure (Continued)

9. Enter the program listed in Figure E15-5.
10. Single step the program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0050	72	SPKwait	Say the phrase
0051	01	(phrase)	
0052	00		
0053	20	BRA	
0054	FB		Do it again
0100	3E	/PA1/	
0101	DB	/H/	
0102	42	/EH1/	
0103	63	/UH3/	No sound
0104	58	/L/	
0105	63	/UH3/	
0106	74	/O2	
0107	B7	/U1/	
0108	3E	/PA1/	
0109	FF	END	

Figure E15-5

11. Change the listing, as indicated by the 01 column in Figure E15-6.

ADDRESS/PITCH	00	01	10	11
0101	1B	5B	9B	DB
0102	02	42	82	C2
0103	23	63	A3	E3
0104	18	58	98	D8
0105	23	63	A3	E3
0106	34	74	B4	F4
0107	37	77	B7	F7

Figure E15-6

12. Single step the program again. Notice that the word sounds flat. It has no inflection.
13. The lists in Figure E15-6 give the phonemes for the word "HELLO." Each column lists a different inflection level. Use this list and select phonemes from different columns in the figure. This will give you different inflections within the word. Enter these into the program and single step the program to determine the effect. Try this several times.

## Discussion

By using different inflection levels, you can greatly affect the sound of the word. You can make it sound more natural, or you can make it sing-song and artificial. It is not enough to simply choose the right phonemes for a word, you must also select the proper inflection. What sounds right to you may not sound right to someone else. It requires a great deal of time and experience before you will be able to select the best sounding phoneme strings.

There are two other controls that affect the way your Robot sounds. These are not dynamic, but are set and not normally adjusted from day to day.

## Procedure (Continued)

14. Use the programming unit to position the arm to the side, away from the back panel.
15. Remove the back panel from your Robot.
16. On the voice synthesis board, locate the Pitch and Volume controls indicated in Figure E15-7.

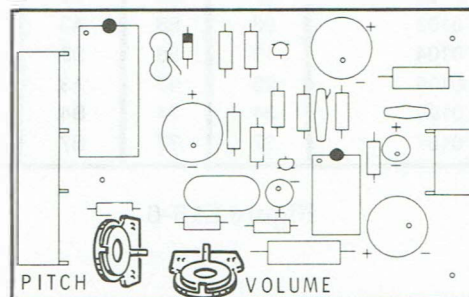


Figure E15-7

17. Execute the program from Figure E15-5.
18. Slowly adjust the Pitch control. Notice the affect it has on the Robot's voice. Leave it at the setting you like best.
19. Turning the Pitch control toward the top of the board  
\_\_\_\_\_ the pitch.  
Raised/Lowered
20. Now slowly adjust the Volume control and notice its affect. Leave it at the setting you like best.
21. Turning the Volume control to the left makes the voice  
\_\_\_\_\_.  
Louder/Softer

## Discussion

In addition to the inflection controls for the pitch of the Robot's voice, there is a control for the general pitch of all phonemes. As you learned in Unit 9, the female voice is generally higher than the male voice. However, the Robot voice was modeled from a male voice spectrum and you cannot make it sound female. This is because raising the fundamental pitch raises the formants proportionally and the formants of the female voice are not proportional to those of the male voice. Therefore, you will only find a small range in the pitch adjustment that sounds right. The volume also affects the speech quality. The Robot's amplifier can be turned up enough to overdrive the speaker. This will distort the voice. If the Robot were bigger, a larger 8 ohm speaker could be used. This would improve the voice quality.





# Experiment 16

## *Robot Language — Motors*

**OBJECTIVES:**

*To state the differences between programs written in native code and those written in Robot language.*

*To decipher Robot language programs comprised of motor control and associated instructions.*

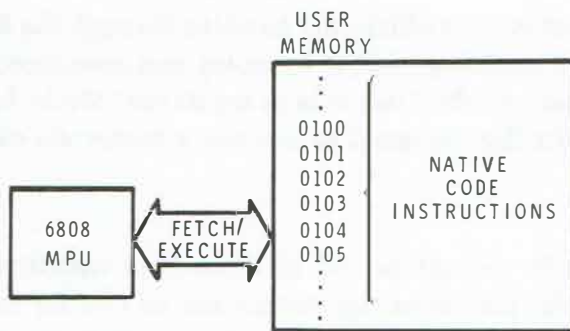
*To write and execute motor control programs using Robot language.*

**Introduction**

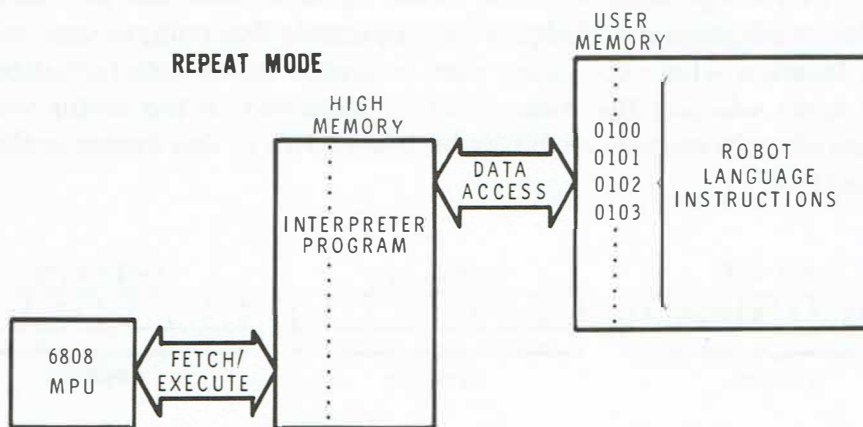
Two modes of program entry are available with your Robot: direct entry (Direct Mode) and Repeat Mode. The first, direct entry, involves machine level programming, where all your instructions are executed at the microprocessor level. This concept is illustrated in Figure E16-1A. Figure E16-1B shows the second programming mode, the Repeat Mode. In this mode, the microprocessor fetches and executes machine code, which is found in ROM interpreter routines, according to your Robot language instructions.

Using the Robot language mode will greatly simplify your programming tasks. For example, three-line Robot language programs can often replace hundreds of machine code instructions. In all, nearly seven thousand machine code instructions reside in your Robot's ROM. The routines made from these instructions form the core of your Robot's interpreter.

Before beginning this experiment, you will need to acquaint yourself with some of interpreter's opcodes and how they are used.

**DIRECT MODE**

**A** MPU FETCHES AND EXECUTES INSTRUCTIONS DIRECTLY FROM YOUR MACHINE CODE PROGRAM

**REPEAT MODE**

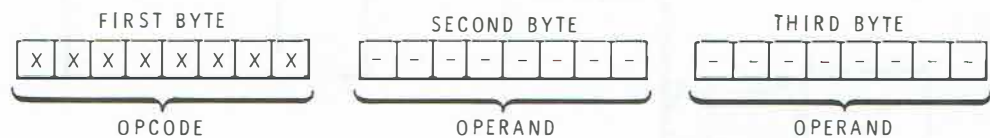
**B** MPU FETCHES AND EXECUTES INSTRUCTIONS FOUND IN INTERPRETER ACCORDING TO ROBOT LANGUAGE PROGRAM

Figure E16-1

## MOTOR CONTROL OPCODES

Motor activity in the Robot is most efficiently handled through the ROM based interpreter. Special opcodes, unique to motor and associated activities, are available whenever the Trainer is in the Repeat Mode. Interpreter instructions specifically designed to operate a motor are called motor control words.

Most motor control words are comprised of three byte instructions. The first byte, the opcode, identifies the instruction as one for motor control. The second and third bytes, the operands, complete the word by either defining the motor direction, speed, and distance directly, or by specifying a memory location where this information can be obtained. Motor opcodes that use immediate addressing have the motor data in the operand. Whereas motor opcodes that use extended or indexed addressing techniques have operands that point to some memory location where the motor data is stored. Motor data formatting is the same whether the motor data is contained in the motor word's operand or in memory. A bit-by-bit breakdown of this format is shown in the box.



Let's take a good look at the different parts of these three bytes, beginning with the opcode byte.

The X's in the first byte represent the opcode. How the interpreter responds to the following bytes depends on this instruction. Basically, there are nine different motor control opcodes. As you will learn, they differ fundamentally by how they control the motor's position, where they get their motor control data, and whether or not other Robot activities are inhibited.

The opcodes for motor motion instructions fall into three categories. The opcodes in the first category specify absolute motor positioning values. Using these opcodes, a motor can be homed to a specific location in its range without regard to its starting position. The six opcodes in this category are:

<u>OPCODE</u>	<u>MNEMONIC</u>	<u>OPERAND</u>
C3	MVWAIT	#(motor, speed, distance)
CC	MVCONT	#(motor, speed, distance)
E3	MVWAIT	address
EC	MVCONT	address
F3	MVWAIT	offset,X
FC	MVCONT	offset,X

The C3 and CC instructions are both immediate mode commands. That is, the two operands following these opcodes contain the necessary motor selection data. These two differ in that the C3 instruction inhibits the interpreter from fetching further instructions during motor movement, while the CC instruction allows the interpreter to continue.

The instructions E3 and EC use the extended address mode to specify the memory location of motor selection data. Instructions F3 and FC operate in a similar manner using the indexed address mode. These four commands are used in programming situations that require computed changes in motor control data.

The second category of motor opcodes specify relative positioning values. Using these instructions, a motor will move to a point relative to its starting position. Opcodes in this category include:

<u>OPCODE</u>	<u>MNEMONIC</u>	<u>OPERAND</u>
D3	MVRELW	#(motor, speed, displacement)
DC	MVRELC	#(motor, speed, displacement)

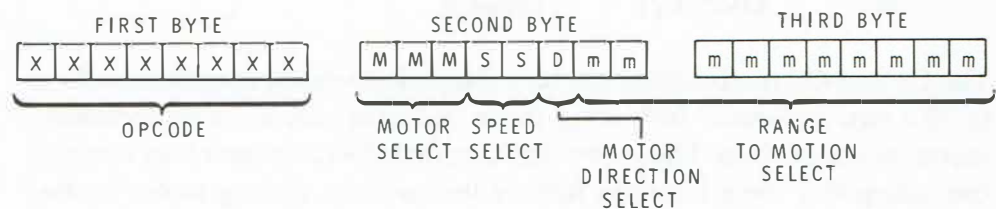
The D3 and DC instructions differ in that the D3 instruction inhibits the interpreter from fetching the next opcode sequence until the motor reaches its destination, just like the C3 opcode mentioned earlier. The DC instruction, like the CC instruction, allows the interpreter to move ahead during motor movement.



The final motor movement category has only one opcode, FD.

<u>OPCODE</u>	<u>MNEMONIC</u>	<u>OPERAND</u>
FD	MVALL	#(extender, shoulder, rotator, pivoter, gripper, head, steering)

Unlike the other motor instructions, FD is eight bytes long. The first byte has the opcode, and the remaining seven bytes are operands specifying the absolute end positions for all the stepper motors. An example of the format for this eight-byte sequence is shown above. You will use it in the remaining experiments. But for now, let's return to our examination of a motor control word.



The first three bits of the second byte of a motor control word (XXXX XXXX MMMS SDmm mmmm mmmm) are used to select the motor. Possible choices for these three bits are:

000	Main Drive
001	Arm Extend/Retract
010	Arm Pivot
011	Wrist Rotate
100	Wrist Pivot
101	Gripper
110	Head
111	Steering

The next two bits of the second byte of the motor control word select the speed (SS) of the motor. Here, possible choices include:

01	Low Speed
10	Medium Speed
11	High Speed

The only exception to these choices is the Robot's shoulder motor, which is only capable of medium and low speed operation.

The next bit of the motor control word, the 'D' bit, selects motor direction for motor opcodes with relative positioning. Here, the choices are best identified by how they effect each motor.

1	0
Drive Backward	Drive Forward
Arm Retract	Arm Extend
Arm Pivot Down	Arm Pivot Up
Wrist Rotate Left	Wrist Rotate Right
Wrist Pivot Up	Wrist Pivot Down
Gripper Close	Gripper Open
Head Left	Head Right
Steer Left	Steer Right

Opcodes like C3, which use absolute positioning, ignore the D bit since, to them, direction control is simply irrelevant.

Finishing the motor control word, the least two significant bits of the second byte and the eight bits of the third byte (mm mmmm mmmm) are used together to determine the range of motor motion. Here is a breakdown of these bits:

<u>MOTOR</u>	<u>RANGE OF MOTION</u>	<u>INITIAL POSITION</u>
Drive Motor	000-3FF	000
Arm Extend/Retract	000-098	000
Arm Pivot	000-086	000
Wrist Rotate	000-093	04D
Wrist Pivot	000-0A5	000
Gripper	000-075	000
Head	000-0BF	062
Steer	000-098	049

### ADDITIONAL OPCODES FOR MOTOR CONTROL

The opcodes listed thus far can really make a difference in generating fast, efficient, motor control programs. You cannot, however, write a complete program using only these commands. One answer is for you to interface your interpreter programs with native code programs. In fact, intermixing program types is often used for more difficult multi-task operations. A better answer is to acquaint yourself with the remaining interpreter opcodes and use them wherever possible.

Your Robot interpreter understands twenty-seven opcodes in addition to the motor control opcodes seen thus far. These opcodes do everything from activating sensors to putting the Robot to sleep. Sixteen of these opcodes control speech, display, and sensor circuits and will be covered in the next experiment in detail. The remaining opcodes are more general in nature and need to be examined here, especially since some of them are essential components to even the most basic program. These opcodes are:

<u>OPCODE</u>	<u>MNEMONIC</u>	<u>MEANING</u>
02	ABORT	Abort base
03	ABORT	Abort steering
04	ABORT	Abort arm
1C	BRBSY	Branch if base busy
1D	BRBSY	Branch if steering busy
1E	BRBSY	Branch if arm busy
21	ZERO	Zero motors then initialize
3A	RTE	Return to Executive Mode
83	EXIT	Exit interpreter
87	SLEEP	Sleep for a prescribed period
8F	PAUSE	Pause for a prescribed period

The ABORT instructions are single byte instructions used to halt motor activity. They can be especially useful in programs where the Robot's senses are used to control motor positioning. For example, your Robot could be programmed to remove small boxes from a conveyor. Should the conveyor stop, the Robot's sonar detector could be used to abort further motor activity.

The BRBSY (branch if busy) instructions are typically used in complex programs where decision making steps are involved. An example program built around the BRBSY instructions might be a self-diagnostic routine operating within a main program whose purpose was to detect a fault in motor I/O logic. A simpler example, used later in this experiment, has the Trainer's voice repeatedly announcing the operation of a motor.

The ZERO instruction (21 ) is used every time you initialize your Robot. When the interpreter receives this instruction, it assumes that its motor positional data is corrupt. Therefore, the interpreter activates ROM subroutines to reposition the motors and update its stored positional data. This single byte instruction is seldom used in programs due to the amount of execution time required.

The RTE (3A ) instruction, a pseudo halt code, is typically the last executable instruction in a Robot language program. This single byte opcode not only prevents a program from running out blindly into memory, but it also returns the Robot to its Executive Mode. The RTE instruction can also be used as a debugging tool to break large complex programs into testable modules.

Not all Robot programs can be written exclusively in Robot language. In fact, it is often desirable to shift a program from the interpreter (Repeat) mode to the native (Program) mode. The EXIT (83 ) instruction does just that; you use it in a Robot language program to drop the interpreter. However, once you are in the native mode, the microprocessor is only able to process standard 6808 instructions. To return to the interpreter, you use the native code instruction SWI (3F ), which provides the necessary software interrupt. Interrupt handling instructions in ROM take care of the rest.

The SLEEP instruction aborts all motion, powers down all subsystems, and powers down the microprocessor. Only the Robot's clock/calender and RAM areas remain active.

The two byte operand for the SLEEP instruction determines the length of the sleep cycle. A minimum code of 00 00 generates a short nap of about 10 seconds, while a maximum operand of FF FF puts the Robot to sleep for about seven and a half days! In order for this instruction to work, you must have previously activated the **SLEEP/NORMAL** switch located atop the experimenter's board.

You can abort your way out of the sleep mode by switching the **SLEEP/NORMAL** switch to the normal position. However, the recovery is not immediate due to a 10 second sleep/time delay. You can also initiate the sleep mode directly from the keyboard by activating the **SLEEP** switch and pressing the **RESET** key. This can prove to be very useful when you wish to speed up battery charge time, but not loose the contents of RAM.

PAUSE instructions (8F ) are usable in just about every type of program. As its name implies, the PAUSE opcode generates a pause in program execution. The two-byte operand following this opcode gives the pause length in 1/16 second intervals. The placement and duration of PAUSE instructions can be very critical to motor control sequences, especially those dealing with Robot steering.

### USING THE MOTOR CONTROL OPCODES

Now that you have had an opportunity to examine a large part of the available Robot language instruction set, you need to see how these instructions can be used. The following procedures will give you an opportunity to follow through a few prewritten programs. Then you can write a few programs of your own.

## Material Required

ET-18 Robot Trainer



## Procedure

1. Ensure that your Robot is fully charged; then remove the charger cable. Turn your Robot ON and initialize its motors.
2. Enter the Repeat program Mode by depressing the A key twice. Then enter the program listed in Figure E16-2, Table A.

ADDRESS	INST/DATA	MNEMONIC OR MEANING
0050	D3	MURELW
0051	C8	
0052	18	
0053	D3	MURELW
0054	D0	
0055	18	
0056	D3	MURELW
0057	D8	
0058	18	
0059	D3	MURELW
005A	48	
005B	44	
005C	D3	MURELW
005D	50	
005E	44	
005F	3A	RTE Return to Executive mode.

Figure E16-2

Table A.

3. Examine your program by pressing first the A key, followed by the **EXAM** key. Make any necessary corrections.
4. Place your Robot within a 6 foot by 6 foot cleared area.



5. Execute your program by pressing the **A** key, the **DO** key and entering the starting address. Observe both the action of the head and arm pivot motors. The head should move to the right in three motions, increasing speed each time. Next, the arm should pivot upward in two motions and at two different speeds.
6. Use the EXAMine mode to change the following inst/data codes:

<u>ADDRESS</u>	<u>PRESENT VALUE</u>	<u>NEW VALUE</u>
0052	18	09
0055	18	09
0058	18	09
005B	44	22
005E	44	22

7. Press the **RESET** key. Then, press keys **3** and **2** to move the head and arm back to their original positions (Utility Mode command 2).
8. Execute the program. Observe the action of the head and arm pivot motors. The head motor will still move in three steps and the arm pivot motor in two, but the total distance of all motions should be half that of Step 5.
9. Use the EXAMine mode to change the following inst/data codes:

<u>ADDRESS</u>	<u>PRESENT VALUE</u>	<u>NEW VALUE</u>
0054	D0	D4
005D	50	54

10. Initialize the head and arm as in Step 7. Then execute the program. Observe how the head and arm pivot motors reverse direction during one of their movements.
11. Use the EXAMine Mode to change the following inst/data codes:

<u>ADDRESS</u>	<u>PRESENT VALUE</u>	<u>NEW VALUE</u>
0052	09	24
0053	D3	72
0054	D4	FA
0055	09	DA
0056	D3	3A

12. Initialize the head and arm as in Step 7. Then execute the program. Observe as well as listen to your Trainer. First the head rotates to the right then the voice announces: "I can turn my head...Ready." The voice routine was added to demonstrate the inhibit function of the D3 opcode. This motor control instruction inhibits all Robot activities during the motor movement cycle.
13. Without INITIALIZING, re-execute the program and observe how the head continues rotating to the right. Execute the program a third time without initializing. Again, the head continues its clockwise rotation.
14. Change the opcode at 0050 to a C3. Initialize the Robot and execute the modified program. Observe how the head rotated to the left instead of the right. Now without INITIALIZING, rerun the program. Note that the head failed to move, yet you know the program executed because of the voice. Try the program a third time; then initialize the head and rerun the program. This exercise illustrated the principle of absolute positioning, a characteristic of the C3 opcode.
15. Change the opcode at 0050 to a CC. Initialize the Robot and execute the program. Observe how the motor moved just as it did with the C3 command. However, this time the voice spoke while the motor was moving. The C3 instruction inhibited the voice, but the CC instruction did not.
16. Change the opcode at 0050 to a DC. Initialize the Robot and execute the program. Run the program a second and a third time without reinitializing. As you can see, the DC opcode controls the motor positioning like the D3 command did; but unlike the D3 opcode, it allows the simultaneous execution of follow-on instructions (the voice routine).

17. Use the EXAMine mode to change or add the following inst/data codes:

<u>ADDRESS</u>	<u>PRESENT VALUE</u>	<u>NEW VALUE</u>
0056	3A	FD
0057	--	00
0058	--	00
0059	--	4D
005A	--	00
005B	--	00
005C	--	62
005D	--	49
005E	--	3A

18. Initialize the head and arm assembly. Then run your new program. Observe how the head automatically returns to its initial position.
19. Re-enter the program in Figure E16-2. Then add the following inst/data codes beginning at address 005F:

FD 00 00 4D 00 00 62 49 3A

20. Run this new program and note how both the head and arm pivot motors return to their initialized positions. This time two motors were initialized by the same inst/data string.
21. Execute your program one last time, but use the Program Mode instead of the Repeat Mode. Observe the results.

## Discussion

In this part of the experiment, you familiarized yourself with the fundamental characteristics of the D3, DC, C3, and CC opcodes. You saw that your ET-18's motors were capable of multiple speeds and that at least two interpreter operations could be performed at the same time.

Once again, references were made to absolute and relative motor positioning. The D3 and DC instructions moved the motors in increments determined by the last part of the motor control word. Repeating the same motor control word simply stepped the motor further and further from its point-of-origin. Figure E16-3A illustrates this concept of relative positioning.

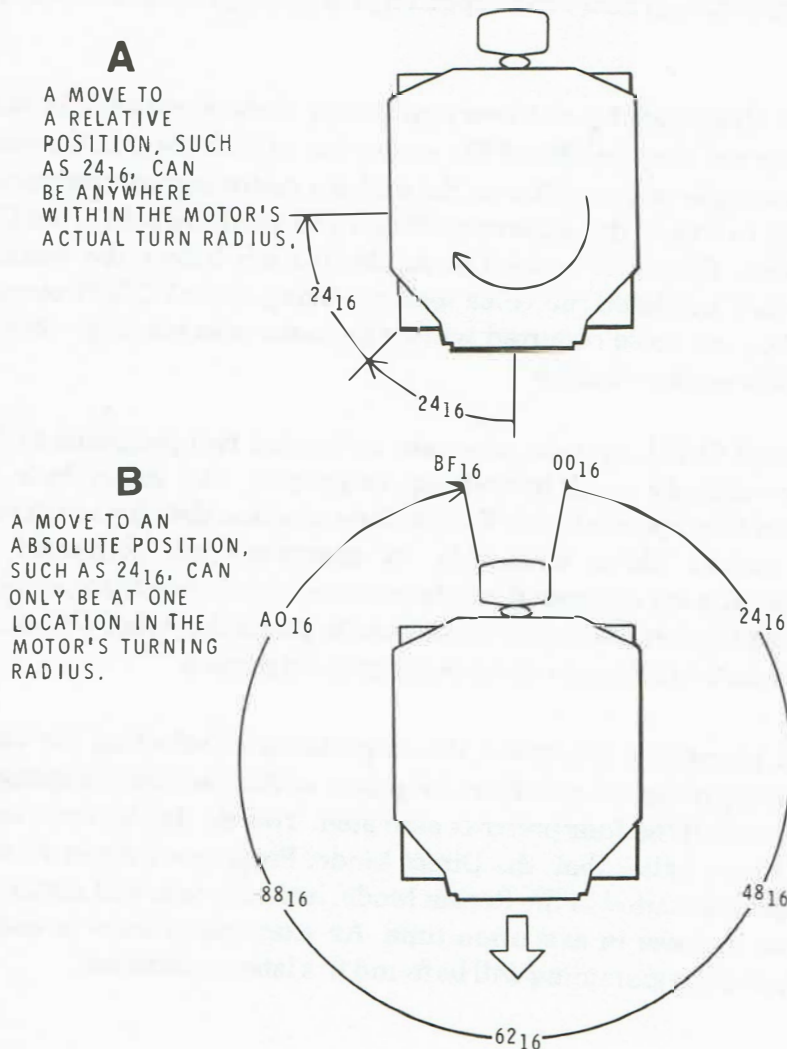


Figure E16-3

Changing the opcodes of the motor control words to C3 or CC caused the head to move in a different direction. This occurred because the C3 and CC opcodes interpret their operands differently. The same bits that indicate travel distance to a relative positioning motor control opcode, indicate an absolute position to these opcodes. In fact, the direction bit is ignored — motor direction is determined by the shortest route to the destination, as shown in Figure E16-3B. Since these two opcodes move motors to absolute physical positions, repeated positioning instructions are also ignored.

A second characteristic of these four motor control opcodes involved whether or not they inhibited the execution of follow-on instructions. Adding a simple voice routine to the end of a motor control sequence allowed you to “hear” the difference. When you used MVWAIT, the D3 or C3 opcodes, the motor moved to its destination before the voice occurred—they inhibited the voice routine. Using the MVCONT opcodes, DC and CC, the voice occurred while the motor was moving—they did not inhibit the voice routine.

The FD, or MVALL opcode, was used in the last two programs to illustrate a commonly used initializing technique. The seven byte long operand of this opcode holds the absolute position data for seven of the Robot’s motors (drive excluded). By executing this command, any motors not already in these absolute positions are moved there automatically. The FD instruction can also be used to preset the Robot’s head, arm, and wrist motors into any other predetermined pattern.

The last procedure illustrated the importance of selecting the correct mode during program execution. Programs written in Robot language can only be used if the interpreter is activated. You do this by entering the Repeat Mode rather than the Direct Mode. Programs written in native code can be executed in the Repeat Mode, however you will notice a tremendous increase in execution time. An example of how to combine both types of programming will be found in a later experiment.

The seven remaining motor control opcodes have unique characteristics that make them especially useful during complex procedures. The BRBSY instructions check the status of certain motors, while the ABORT instructions stop motors in operation. The E3 and F3 motor control opcodes control motors and follow-on instructions just like C3. In the same regard, EC and FC opcodes operate like CC. What makes these four motor control opcodes unique is how they obtain their motor control data. Rather than the immediate addressing mode used by CC, C3, DC and D3, these opcodes read their data from memory through either extended or indexed addressing.

Before you begin writing your own programs, you need to practice deciphering the opcodes and motor control words discussed thus far.

## Procedure (Continued)

22. Interpret the following motor control words:

A. D3 00 6A

---

---

B. CC D0 20

---

---

C. DC 7C 93

---

---

D. C3 3C 25

---

---

E. FC 02 2A

---

---

F. FD 49 86 42

A5 75 62 49

---

---

---



23. Check your interpretations with the ones listed below. If you need help, refer back to the introduction section of this experiment.
- A. D3 00 6A      Move the Drive forward 6A increments at slow speed. Use relative positioning and finish the move before continuing.
  - B. CC D0 20      Rotate the head to absolute position 20 at medium speed. Execute a follow on command if possible.
  - C. DC 7C 93      Rotate wrist left to a position 93 increments relative to the starting point, at fast speed. Execute a follow-on instruction if possible.
  - D. C3 3C 25      Extend or retract arm to absolute position 25 at fast speed. Finish the move before continuing.
  - E. FC 02 2A      Go to address 022A and retrieve absolute mode motor data. Execute a follow on instruction if possible.
  - F. FD 49 86 42  
A5 75 62 49      Make the following absolute moves: extender to 49, arm pivot to 86, wrist rotator to 42, wrist pivoter to 75, head to 62 and steering to 49.
24. Write a program using Robot language that moves the head to its full left stop, to its full right stop, and then back to its initialized position. Use slow speed for the first move, medium speed for the second move, and fast speed for the last move.
25. Enter your program starting at address 0050. Remember to EXAMine your program before continuing.

26. Execute your program in the Repeat Mode and observe the results. Now compare your program with the one shown in Figure E16-4. Your program may vary from this one; what is important is that it accomplish the stated goals.

ADDRESS	INST/DATA	MNEMONIC OR MEANING
0050	C3	MVWAIT
0051	C8	
0052	00	
0053	C3	MVWAIT
0054	D0	
0055	BF	
0056	C3	MVWAIT
0057	D8	
0058	62	
0059	3A	RTE

Figure E16-4

27. Write a program using Robot language that does the following:

- A. Moves platform forward 20 increments.
- B. Moves arm to left side of Robot.
- C. Opens gripper fully.
- D. Extends arm to full stop position.
- E. Closes gripper half way.
- F. Retracts arm
- G. Moves arm to front left of Robot.
- H. Extends arm to full stop position.
- I. Opens gripper fully.
- J. Retracts arm.
- K. Moves arm to left side of Robot.
- L. Repeats steps D through K till Aborted.

NOTE: Use slow speed for the platform, medium speed for steps D through J, and high speed for the rest. Use a BRA instruction for the loop.

28. Enter your program starting at address 0050. Remember to EXAMine your program before continuing.

29. Execute your program in the Repeat Mode and observe the results. Now compare your program with the one shown in Figure E16-5. As before, your program may vary from the one in the Figure. It must, however, complete the stated goals.

ADDRESS	INST/DATA	MNEMONIC OR MEANING	
0050	21	ZERO	Initialize motors.
0051	D3	MVRELW	
0052	08		Drive forward slow 20.
0053	20		
0054	C3	MVWAIT	
0055	D8		Move arm to 96, fast.
0056	96		
0057	C3	MVWAIT	
0058	B8		Open gripper to 75, fast.
0059	75		
005A	C3	MVWAIT	
005B	28		Extend arm full, slow.
005C	98		
005D	C3	MVWAIT	
005E	A8		Close gripper halfway, slow.
005F	35		
0060	C3	MVWAIT	
0061	28		Retract arm, slow.
0062	00		
0063	C3	MVWAIT	
0064	C8		Move arm to front left, slow.
0065	BF		
0066	C3	MVWAIT	
0067	28		Extend arm full, slow.
0068	98		
0069	C3	MVWAIT	
006A	A8		Open gripper to 75, slow.
006B	75		
006C	C3	MVWAIT	
006D	28		Retract arm, slow.
006E	00		
006F	C3	MVWAIT	
0070	D8		Move arm to 96, fast.
0071	96		
0072	20	BRA	Branch back to 005A.
0073	E6		
0074	3A	RTE	

Figure E16-5

## Discussion

You have now had an opportunity to write a couple of basic motor movement programs. Before getting into more involved routines, you will need to know the rest of the interpreter's special opcodes. These will be covered in the next experiment.

UNIT TWELVE

UNIT TWELVE



# Experiment 17

## *Robot Language — Sensors And Sound*



**OBJECTIVES:**

*To decipher Robot language programs comprised of sensor and sound control instructions.*

*To write and execute sensor and sound control programs using Robot language.*

*To write programs that switch between the interpretive and native code modes.*

**Introduction**

In the last Experiment, you began to acquaint yourself with the ET-18's Robot language. You examined the different types of opcodes found in basic motor control programs. Afterwards, you learned how to write goal-oriented motor control words and then link them together to perform repetitive tasks. In this experiment, you will examine the remaining Robot language instructions and learn how to use them to build complex programs incorporating sensor and speech functions. You will also see how to design programs that operate in both the interpretive and native modes.

**SENSOR CONTROL OPCODES**

When you are developing robot programs, you often need to use one or more of the Robot's sense functions. You were acquainted with the basic functions of these onboard senses back in Experiment 13. Unlike programs dedicated strictly to motor control, programs using sensors often have to use a mixture of interpreter and native code.

Sensing routines usually require a combination of interpreter and native code instructions. Since sensing activities tend to be very time sensitive, simply executing native code routines through the interpreter, a procedure often used with simple motor programs, is generally unacceptable. Instead, motor/sensor programs need to shift between interpreter and native code modes. This way, they can take full advantage of the interpreter's programming simplicity, yet still retain native code speed during critical sensor routines.

Basically, there are two types of Robot language opcodes for sensor control. Together, they embody the ENABLE and DISABLE instructions listed below.

<u>OPCODE</u>	<u>MNEMONIC</u>	<u>MEANING</u>
41	ENEYE	Enable the eye sensor
42	ENEAR	Enable the ear sensor
45	ENSON	Enable sonar
4B	ENMOT	Enable motion detector
51	DISEYE	Disable the eye sensor
52	DISEAR	Disable the ear sensor
55	DISSON	Disable sonar
5B	DISMOT	Disable motion detector

Basic control of Hero's sensors is affected either through enable/disable logic signals or by applying/removing subassembly power. Whatever the actual requirements, all hardware switching is done automatically by the interpreter program. All you need do is supply the appropriate enable or disable opcode.

Your Robot has five senses in all. It can sense time, light, sound, motion, and the distance to an object. These sensors can be used alone or in concert with other Robot functions, including other sensors. You will find it extremely difficult to program your Robot for a really sophisticated task without incorporating at least one sense detector.

## REVIEWING THE SENSES

Your Robot has a built-in clock and calender circuit that can be easily interfaced into control programs requiring time determinations. The clock circuit in your Trainer is a type often referred to as a real-time clock. It is software independent; therefore, it can operate even when the Robot is turned off. The time and date information is available to your programs through special permanent utility programs. You can obtain time or date information by accessing memory locations used as storage by these routines.

Later in the procedure section of this experiment, you will learn how to set and read the time and calendar functions of your Robot.

To detect different light intensities, the Robot uses an analog to digital converter driven by a sensitive analog amplifier. The actual sense component is a light-sensitive resistor, part of a voltage divider in the amplifier. The eight-bit output of the A/D converter can be read directly at address C240 and is capable of indicating 256 different light intensities.

As the light sensor uses direct I/O, it can remain enabled throughout a program whether its output is being read or not. The only exception to this is when you wish to also use the sound detector. Both sensors use the same A/D converter and I/O port address. Therefore, these senses should not be enabled simultaneously. They could, however, be toggled back and forth through software control. The ENEYE instruction is used to enable this sensor and the DISEYE is used to disable it.

The sound detector in your Robot indicates the intensity of surrounding noise. As we mentioned earlier, it shares its A/D circuitry and I/O port with the light sensor. Therefore, it too can indicate 256 different output levels. The ENEAR opcode enables this sensor and the DISEAR opcode disables it.

The motion detector is an ultrasonic device, much like those used in intrusion alarm systems. As such, its primary purpose is to detect motion in the Robot's immediate vicinity. When activated by the ENMOT instruction, the motion sensor has a direct connection to the microprocessor's interrupt circuitry. When motion is detected, the  $\overline{\text{IRQ}}$  interrupt it generates halts programs in execution and changes the program counter to 0027. You can activate special routines by placing a jump instruction at this address with the jump operands at 0028 and 0029. Later, you can use the RTI (return from interrupt) instruction to return the microprocessor to the original program. The DISMOT opcode disables this sensor.

The last sense of your Robot is its sonar range finder. When activated by the ENSON instruction, the sense's transmitter sends out ultrasonic pulses. Any nearby object would cause some of this transmitted signal to bounce back to the Robot, where an ultrasonic receiver awaits. Distance is easily determined by the time it takes an echo to reach the receiver. Direction can be determined from the Robot's head position due to the narrow dispersion characteristics of the ultrasonic transducers.

The sonar timer circuit, part of the I/O board, measures the time interval between the leading edge of the transmitter pulse and the leading edge of the received echo. This interval is a measure of the range to the object causing the echo. If no echo pulse is received within approximately 500 milliseconds after the transmit pulse, the timer circuit resets itself and awaits for the next transmit pulse cycle.

When an echo returns from an object within the range of the system, a binary counter stops and holds its count. The received echo is also used to generate an IRQ interrupt. In servicing the interrupt, the CPU reads the counter through port address C220 and places this value at address 0011. Then the CPU resets the sonar circuit for another cycle of operation.

Once enabled, the sonar circuit operates independent of any executing programs. Interrupts and data updates are handled automatically, as are the necessary return from interrupt (RTI) instructions. The DISSON instruction disables the sonar sensor.

## USING THE SENSORS

Now that you have refamiliarized yourself with the basic operation of your Robot sensor circuits and have learned their respective Robot language opcodes, you need to see how these sensors can be used. The following procedures will give you an opportunity to not only follow through the operation of some prewritten programs, but to also write a few of your own.

## Material Required

ET-18 Robot Trainer

## Procedure

1. Turn your Robot On. Press the **3** key to enter the Utility Mode. Then press the **5** key to access the clock function. The display should read:

**HH\_\_SS**

2. Enter the hour by pressing two numbers, using either 12 or 24-hour notation. For example, one o'clock PM could be entered as either 01 or 13. The two left-most displays should now reflect your entry.
3. Enter the minutes digits. The two center displays should reflect these entries.
4. Enter the seconds digits. When you enter the second digit, the display will change to:

**A P 24**

The 'A' display corresponds to AM, the 'P' display to PM and the '24' display to 24-hour format.

5. Press the **D** key if you want your entered time to be 'AM', the **E** key for PM, or the **F** key for 24-hour format.

Note: Your Robot's clock is now running and will continue running even with the power turned off.

6. Press the **3** key to re-enter the Utility Mode. Then press the **6** key to activate the date set function. The display should indicate:

**yy\_\_dd**

7. Enter two digits for the year, two digits for the month, and two digits for the day. For example August 20, 1982 would be entered by pressing "820820". This date is inserted into memory upon your entry of the last digit.
8. Press the **3** key (Utility Mode) and the **7** key to display the time. Note the time on the display: hours to the left, then minutes, and finally seconds (there is no AM/PM indicator on this display). The time display will remain active until you press the **RESET** key.
9. Press **RESET**. Then press the **3** and **8** keys to display the date. This display also remains on until you press **RESET**.



## Discussion

These procedures showed you how to enter time and date information into your Robot's on-board clock, and then retrieve it on command. Ports C2C0 and C300 are used by your Robot's utility programs to access this clock. Through careful programming, you can also access the clock and then use its output to control your programs.

Time and date information can only be read from the clock display port, C300, one nibble at a time. You can request any nibble of the time/date data by sending the appropriate enabling nibble to clock port C2C0. Figure E17-1 shows the various enabling bit combinations and their corresponding time/date characters.

OUTPUT TO PORT C2C0	TIME/DATE CHARACTERS RETURNED FROM PORT C300 (BITS 0-3)
00	Seconds digit, least significant (0-9)
01	Seconds digit, most significant (0-5)
02	Minutes digit, least significant (0-9)
03	Minutes digit, most significant (0-5)
04	Hours digit, least significant (0-9)
05	Hours digit, most significant hours use bits 0-1 AM/PM in bit 2 12 hr/24 hr in bit 3
06	*Day of week (0-6)
07	Day, least significant (0-9)
08	Day, most significant (0-3)
09	Month, least significant (0-9)
0A	Month, most significant (0-1)
0B	Year, least significant (0-9)
0C	Year, most significant (0-9)

\*The day of week is not entered through the Utility Routines. It could be used through direct software programming.

Figure E17-1



## Procedure (Continued)

10. In the space below, record your Robot's present time/date information. Use the Utility Mode commands from the previous procedures.

HOURS	MINUTES	SECONDS	YEAR	MONTH	DAY

11. Enter the EXAMine Mode and check the contents of addresses C2C0 and C300. Record the information.
12. Using the following table, change the contents of address C2C0. Then record the contents of C300. Repeat this change and record process until completed. (Use EXAMine)

PLACE THIS VALUE AT ADDRESS C2C0		RECORD VALUE AT ADDRESS C300
1	0C	_____
2	0B	_____
3	0A	_____
4	09	_____
5	08	_____
6	07	_____
7	06	_____
8	05	_____
9	04	_____
10	03	_____
11	02	_____
12	01	_____
13	00	_____

13. Write down the least significant hexadecimal bits from column two of Step 12 to column two below:

VALUE AT ADDRESS C2C0			VALUE AT ADDRESS C300 (Bits 0-3 only)
1	0C	[ MS Year ]	___
2	0B	[ LS Year ]	___
3	0A	[ MS Month ]	___
4	09	[ LS Month ]	___
5	08	[ MS Day ]	___
6	07	[ LS Day ]	___
7	06	[ Day of Week ]	___
8	05	[ MS Hour ]	___
9	04	[ LS Hour ]	___
10	03	[ MS Minute ]	___
11	02	[ LS Minute ]	___
12	01	[ MS Second ]	___
13	00	[ LS Second ]	___

## Discussion

Except in the case of the most significant hour digit, you should recognize the least significant hex bits from port C300 as Year, Month, Day, Day of Week, Hour, Minute and Seconds. The most significant hour digit may be difficult to recognize since only the least two bits (0-1) of this hex bit are used to determine the hour digit. Bits 2 and 3 contain AM/PM and 12/24 hour format information. So, if you ignore bits 2 and 3, you have the correct, most significant hour digit. If bit 3 is low, your clock is set to display time in the standard 12-hour format. If it is high, your clock's output uses the military style 24-hour format. Bit 2, useful only in the 12 hour format, determines AM (low) or PM (high).

## Procedure (Continued)

14. Enter the program shown in Figure E17-2. Check your entries using the EXAMine Mode. Note the different technique used to list the program steps. Rather than write simply by addresses, here, the program is organized by executable instructions. Most programmers write their program listings using this technique.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0060	CE 00 40	LDX 0040	Start time storage at 0040 <sub>16</sub> .
0063	86 0C	LDA 0C	Highest request value.
0065	B7 C2 C0	STA C2C0	Send request to clock port.
0068	36	PSHA	Save A on stack.
0069	B6 C3 00	LDA C300	Read clock data.
006C	A7 00	STAA 00	Store value (no offset).
006E	08	INX	Increment storage address.
006F	32	PULA	Get A from stack.
0070	4A	DECA	Decrement A for next request value.
0071	2C F2	BGE	Branch if $\geq 0$ .
0073	3E	HALT	

Figure E17-2

Routine to read time/calendar data from clock port into memory.

15. Run the program in the Program Mode.
16. After a moment or two, hit **RESET**. Then record the contents of the address below:

<u>ADDRESS</u>	<u>CONTENTS</u>	<u>ADDRESS</u>	<u>CONTENTS</u>
0040		0047	
0041		0048	
0042		0049	
0043		004A	
0044		004B	
0045		004C	
0046			

17. Hit the **RESET** key. Now check your Robot's time and date information using the Utility Mode. Compare this date with what you entered in Step 16.

## Discussion

The program in Figure 2 reads the clock's time and date information from port C300 and places this information in memory. Once in memory, you can easily compare it with a time reference stored in a program. Thus, you can design programs to activate other routines at a prescribed time of day, time of week, time of month, or even time of year.

Ports C2C0 and C300, used here to access time and date information, have other functions addressed through their four higher order bits (4-7). Exact information on these ports can be found in your Robot's technical manual.

Now let's look at some programs that use some of the other Robot senses.

## Procedure (Continued)

18. Enter the Repeat program Mode. Then enter the program listed in Figure E17-3. Check your program.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0100	41	ENEYE	Exit interpreter.
0101	83	EXIT	
0102	BD F6 4E	JSR REDIS	
0105	B6 C2 40	JSR SENSEPORT	
0108	BD F7 AD	JSR OUTBYT	
010B	CE 28 00	LDX 28 00	Load X register.
010E	09	DEX	Decrement X.
010F	26 FD	BNE FD	Branch back if not zero.
0111	20 EF	BRA EF	Branch always.
0113	3E	HALT	

Figure E17-3

Program to display the output from either the light or sound sensor.

19. Run the program. Move your hand back and forth in front of the Robot's light sensor. Observe the display indications as it varies with the position of your hand. Cup your hand over the light sensor and note the display indication. Finally, place a bright light near the sensor to see how it affects the display reading. If you have access to the A/D board, repeat these tests and note the binary changes of the eight LED's.

20. Change the inst/data code at address 0100 to a 42 (ENEAR). The program will now check for ambient levels of background noise instead of light.
21. Make your experiment area as quiet as possible. Then run the modified program. Set the sound sensitivity adjustment on the A/D board to get a low display reading. Observe the display changes as you talk to your Robot. Move around the room and make different types of noises. If they are visible, note the binary changes of the eight LEDs on the A/D board.

## Discussion

This program demonstrates two senses, light and sound. Probably the single most important part of this program was the call to SENSEPORT, at address C240. One call to this subroutine immediately places the current output value of the A/D converter in accumulator A. Once this is accomplished, you can use this value to control conditional branches to other subroutines; or in this case, simply display the A/D's output value. The other two ROM subroutines used cleared the displays (REDIS at F64E) and then displayed the sensor value in accumulator A (OUTBYT at F7AD).

## Procedure (Continued)

22. Enter the program shown in Figure E17-4. Check your program in the EXAMine Mode.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0200	4B	ENMOT	Enable motion detector.
0201	83	EXIT	Exit interpreter mode.
0202	0E	CLI	Clear interrupts.
0203	86 7E	LDAA 7E	Load accumulator A.
0205	97 27	STAA 27	Store at address 0027.
0207	CE 02 0E	LDX 020E	
020A	DF 28	STX 28	Store at addresses 0028, 0029.
020C	20 FE	BRA FE	
020E	3F	SWI	Return to interpreter.
020F	5B	DISMOT	Turn off motion detector.
0210	72 FB 56	SPKWA FB 56	Speak phonemes beginning FB 56.
0213	7E 02 00	JMP 02 00	
0216	3A		

Figure E17-4

A simple intrusion alarm program.



23. Run the program. Make a motion in front of the motion detector and “listen” to the results. Move further away from your Robot. Then move into the path of the sensor. If possible, try to locate this sensor’s maximum range.

## Discussion

This program illustrated the capabilities of the Robot’s motion detector. The simple sound routine was used only to show how motion can be used to trigger a routine. This sensor does not have a direct port connection like the clock, light, or sound sensors. Instead, it generates an interrupt which must be serviced by the microprocessor directly. Linking to another program is done by placing a Jump instruction at the interrupt vector, address 0027, followed by the program’s address at addresses 0028 and 0029. With this technique, you can link into any type of program. You must remember to disable the motion detector as soon as you leave the sensor loop routine. Otherwise, the motion detector continues to generate interrupts which will interrupt your follow-on program.

## Procedure (Continued)

24. Enter the program shown in Figure E17-5. Check your entries.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0300	CC D7 BE	MVCONT D7 BE	Move head to BE.
0303	8D 07	BSR 07	Branch to subroutine.
0305	CC D7 02	MVCONT D7 02	Move head to 02.
0308	8D 02	BSR 02	Branch to subroutine.
030A	20 F4	BRA F4	Branch to beginning.
030C	45	ENSON	Enable sonar.
030D	83	EXIT	Exit interpreter.
030E	96 11	LDAA SNRRNG	Get range.
0310	BD F6 4E	JSR REDIS	Clear displays.
0313	BD F7 AD	JSR OUTBYT	Display range.
0316	4F	CLRA	Clear accumulator.
0317	BD F7 C8	JSR OUTCH	Display empty digit.
031A	BD F7 C8	JSR OUTCH	Display another empty digit.
031D	96 10	LDAA SNRHIT	Get hit values.
031F	BD F7 AD	JSR OUTBYT	Displays hits.
0322	3F	SWI	Return to interpreter.
0323	1E E8	BRBA	Branch if arm (head) is busy.
0325	39	RTS	Return.

Figure E17-5



25. Run the program and observe your Robot's actions closely. The head should immediately begin swinging between absolute positions 02 and BE. During the head's motion, the LED displays indicate sonar activity. The displays on the left indicate the number of "hits" recorded, while the displays on the right indicate the range to the objects.
26. Move into the Robot's range of motion and note the change in the range display as the Robot's head scans your location. Move closer to the Robot and note the change in range indication. Try and determine the sonar's minimum range. This will be the point where the display becomes very erratic.

## Discussion

This last program not only showed you how to activate the sonar sensor, but also how to retrieve the hit and distance to target information. Once the sonar detector becomes active, address 0010 stores the number of sonar hits, while address 0011 stores their range. The display routines used in this program simply showed how easy it was to retrieve this information.

## Speech Routine Discussion

Your Robot's interpreter and ROM routines provide everything you'll need to produce phoneme speech. In fact, there are even 28 ready-made phrases. The following interpreter opcodes are used for speech control:

05	ABORT	Abort Speech
1F	BRBSY	Branch if busy
61	SPKCX	Speak continue, indexed addressing
62	SPKWX	Speak wait, indexed addressing
71	SPKCA	Speak continue, extended addressing
72	SPKWA	Speak wait, extended addressing

The first two of these opcodes, ABORT and BRBSY, are similar in operation to those used for the motors and senses. Abort is used to stop a speech instruction. BRBSY is used to see if a speech operation is still functioning and, if so, branch to another address.

The remaining four opcodes start the speech interpreter, a specialty routine designed to read data structures called "phoneme trees." During the phoneme tree read process, the interpreter takes a phoneme and outputs it to the Votrax SC-01 synthesizer chip. It then waits while the synthesizer generates the appropriate sound. When done, the synthesizer sends back a "request" signal, to which the speech interpreter outputs the next phoneme. This process continues until all the phonemes in the tree are exhausted or an ABORT speech instruction is received.

The phoneme tree is the operand to a speak instruction. Unlike operands for other opcodes, a speech operand can have any number of bytes. This is why only extended or indexed address modes are used.

A phoneme tree is comprised of a string of phoneme codes ended by one of two special stop codes, an BF or FF. The BF stop code is actually a branch-to code and is always followed by the two byte address of another phoneme string. The FF stop code causes the interpreter to return to a previous phoneme tree. If the current phoneme tree is the top tree level, the speech process ends.

To speak a word, phrase, or sentence, you simply list phonemes and any branch instruction (BF,xx) in sequential memory locations and terminate the list with an FF code. Before you can call this particular tree, you must also generate a phoneme tree for each branch instruction used. Then you invoke one of the speak opcodes along with the necessary address pointer. This pointer tells the interpreter the starting address of the phoneme tree.

As we already mentioned, you have the choice of four different speech opcodes. If you want to prevent any other Robot activities from beginning during a speech process, you would use either the SPKWX or SPKWA instruction. SPKWX uses index addressing to locate its phoneme tree, while SPKWA uses extended addressing. If you would like to begin some other Robot activity during the speech process, you would use either the SPKCX or SPKCA instruction. The first uses indexed addressing and the latter uses extended.

Since you already practiced concatenating phoneme strings in an earlier experiment, the emphasis in these final procedures will be to familiarize you with addressing techniques and phoneme tree branching.

## Procedure

27. Enter the Repeat program Mode and enter the program shown in Figure E17-6. Check your program.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0100	72 FA DA	SPKWA FADA	Speak phoneme tree at address FADA.
0103	C3 D8 00	MVWAIT D8 00	Move head to absolute position 00.
0106	3A		

Figure E17-6

28. What will this program do? \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_
29. Run the program. Observe and listen. Then compare what the program did to what you wrote in Step 28. Initialize the head using Utility command 3,2.
30. Change the opcode at address 0100 to a 71. Rerun the program. What is different? \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_
31. Change the opcode at address 0100 to a 61. Initialize the head as in Step 29. Then rerun the program. What happens? \_\_\_\_\_  
 \_\_\_\_\_  
 Why? \_\_\_\_\_
32. Write a short program using the SPKCX opcode (61). Have the Robot speak the phoneme tree starting at address FAAA while it moves its arm pivot to its full upward stop position.
33. Initialize the Robot's head, enter your program, and then run it.

34. Compare your program to that shown in Figure E17-7. Could you have shortened your program? \_\_\_\_\_. If so, how? \_\_\_\_\_.
35. Enter the program shown in Figure E17-7. Have the Robot speak the phoneme tree at address FAC0 without changing the index address at 0101 and 0102. Initialize the arm pivot motor. Then run your altered program. Did it speak the phrase listed for address FAC0 in Table E17-1? \_\_\_\_\_. You should have changed the offset byte at address 0104 from 00 to 16, the difference between FAC0 and FAAA.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0100	CE FAAA	LDX FAAA	Load index with FAAA.
0103	61 00	SPKCX 00	Speak phonemes at address in index register.
0105	C3 50 86	MVWAIT 50 86	Move arm pivot to 86.
0108	3A	RTE	Return to Executive Mode.

Figure E17-7

Routine to speak phonemes using indexed addressing mode.

36. Initialize the arm again. Then enter the program listed in Figure E17-8. Check your entries.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0100	72 01 03	SPKWA 01 03	Pause for 2 seconds.
0103	BF FA 4B		Move drive forward 2F.
0106	BF FB C1		Branch 05 if drive busy.
0109	BF FC 5B	RTE	Speak phoneme tree at 0110.
010C	BF FE 8C		Branch to beginning.
010F	FF		Speak phoneme string at 0100.
0110	3A		Branch back to BRBSY test.
			Return to Executive mode.

Figure E17-8

Program to demonstrate phoneme tree branching techniques.

37. Which phrases will the Robot speak? (See the table in Figure E17-9.) \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_
38. Run the program and compare your answers in Step 37 to what you hear.

PHONEME TREE ADDRESS	PHONEME TREE CONTENTS
FA4B	Hello, my name is Hero.
FA64	Hello. I am Hero, The Heath Educational Robot.
FA93	I can talk, like this.
FAAA	I can move my arm.
FAC0	I can use my gripper.
FADA	I can turn my head.
FAF1	And I can move about.
FB09	Also, I can sense light, sound, and movement.
FB37	I see light.
FB45	Wow! It is dark.
FB56	Wait! Something moved.
FB6B	I hear that!
FB7B	There is something in my way.
FB9A	Help! Help! Help! Alarm, Emergency!
FBC1	I have a brain, just as you do. But my brain is a computer. My owner programs my computer for me and I always do as I'm programmed.
FC5B	There is no such thing as a bad Robot, just a misprogrammed one.
FC9D	Gosh, I think I'm just about perfect!
FCC5	I think you are cute. Give me a HUG, Robots need love too.
FD04	You are very attractive for a human.
FD2C	Yes sir, you are handsome.
FD46	Your wish is my command.
FD60	Oh my! Please do not do anything to hurt me!
FD91	Please be quiet, I'm trying to sleep!
FDBA	I think I make an excellent pet. I'm even house trained.
FDF7	People stare at me a lot. I suppose it's because I'm so short.
FE33	Warning! Warning! Intruder. I have summoned the police.
FE6D	Oh no! I don't do windows!
FE8C	I am incapable of making a mistake. Therefore, you must be wrong.

Figure E17-9

## Discussion

These examples have shown you how easy it is to program your Robot for speech. By far, generating phoneme trees will always prove to be the most difficult part of the entire speech process. When using speech, you will find it to your advantage to place your phoneme trees above your main program. This way, you can edit their phonemes without disturbing the contents of the main program.

## PROGRAMMING MOTION, SENSORS, AND SPEECH

Your Robot Trainer is capable of performing some rather amazing tasks. From announcing the time when you approach it to finding its way out of a complex maze, your Robot's power is limited more by your imagination than by any physical characteristics. You should study the various programming examples in the Robot's Users Guide and Technical Manual. Use these programs as examples of what has been done. Outline yourself some simple but interesting Robot tasks it can do; then put Hero to work completing these tasks. The more programs you write, the better you will become at writing them and the more interesting you will be able to make them.



12-248

The first part of the document is a letter from the President of the United States to the Congress. The letter is dated January 1, 1863, and is addressed to the House of Representatives. The President discusses the state of the Union and the progress of the war against the Confederacy. He mentions the Emancipation Proclamation and the importance of the Union's victory.

UNIT TWELVE

The second part of the document is a report from the Secretary of the War Department. The report is dated January 1, 1863, and is addressed to the President. The Secretary discusses the military operations of the Union Army and the progress of the war. He mentions the Battle of Gettysburg and the importance of the Union's victory.

UNIT TWELVE

# Experiment 18

## *Cassette and Teaching Pendant Interface*

**OBJECTIVES:**

*To download programs from memory to a cassette tape.*

*To upload programs from cassette tape to memory.*

*To use the teaching pendant in the Manual Mode to control the Robot's eight axes of motion.*

*To program the Robot's motor functions in the Learn Mode using the teaching pendant.*

*To execute motor control programs entered with the teaching pendant.*

*To modify programs originally entered with the teaching pendant.*

## **Introduction**

The cassette I/O utilities and the Teaching Pendant control utilities are two very important features of your Trainer. The two programming utilities, Manual and Repeat, let you practice moving the Robot through a task, and then generate and repeat a program, all using a special remote control handle—called a teaching pendant. Two cassette utilities let you store and later retrieve programs and data from your Robot's memory. Thus, eliminating the need to ever key or pendant entry a program more than once.

## CASSETTE I/O

The Robot Trainer has a cassette input/output interface and a ROM based upload/download controller program. Together, they let you store programs from the RAM area of your Robot's memory on inexpensive cassette tape. Once on tape, these same programs can be read back into memory and re-executed. Dumping a program from memory to tape is called "downloading." Reading from the tape back into memory is called "uploading."

Nonvolatile program storage, such as that provided by your Trainer's cassette I/O facility, is especially important when a robot is to be used on a daily basis performing long and complex programs. Tape storage also aids in program development by allowing the storage of vast libraries of reusable subroutines.

The cassette I/O utilities located in ROM are directly accessible through the Executive monitor routine. They are extremely easy to use, as you will see through the following procedures.

## Procedure

1. Ensure that your Robot is fully charged. Then disconnect the charger cable.
2. Turn your Robot on and enter the program listed in Figure E18-1, using the Repeat Program Mode. Examine your program before continuing.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0050	8F 00 20	Pause 0020	Pause for 2 seconds.
0053	CC 08 2F	MVCI 08 2F	Move drive forward 2F.
0056	1C 05	BRBSY 05	Branch 05 if drive busy.
0058	72 01 10	SPKWA 01 10	Speak phoneme tree at 0110.
005B	20 F3	BRA F3	Branch to beginning.
005D	72 01 00	SPKWA 0100	Speak phoneme string at 0100.
0060	20 F4	BRA F4	Branch back to BRBSY test.
0062	A3	RTE	Return to Executive mode.
0100	0C 37 37 1D 27 14 03 03 03 03 FF		
0110	9F AA 64 63 65 2A 03 03 FF		

Figure E18-1

3. Execute the program and observe the results.
4. Without turning the Trainer off, connect a cassette recorder to the Trainer's rear panel audio input/output jacks. Use Figure E18-2 as a guide.

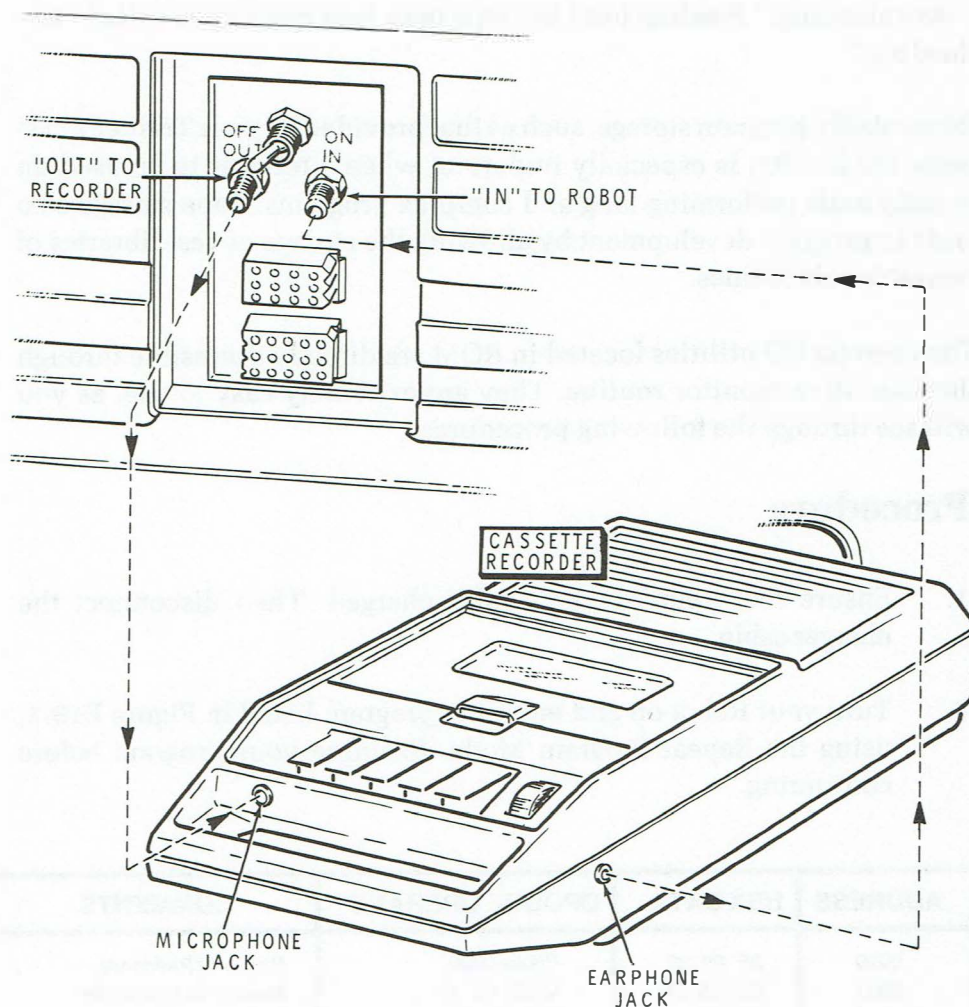


Figure E18-2

5. If Hero is not already in the Executive mode, press the **RESET** key.
6. Press the **3** key to enter the Utility Mode and **3** again to begin the "download" routine. The display should show:  
    \_\_\_\_ Fr.
7. Enter the memory address of the beginning of the program. Now the display should show:  
    \_\_\_\_ La.
8. Place a cassette tape into the cassette. Then rewind the tape as necessary. Note the value of the cassette's digital counter (you may reset the counter to zeros if you wish).
9. Place the cassette recorder into its Record mode. Then enter the final address of the program. The program will now download to the recorder. When the program is completely recorded, the Robot will automatically re-enter the Executive Mode. Stop the recorder at this time.

NOTE: You may wish to let the recorder run for a few moments before actually turning it off. The blank space left on the tape can be very helpful in separating later programs.

10. Re-execute the program already in memory and, once again, observe the results. As you can see, dumping a program to tape does not change the contents of your Trainer's memory.
11. Turn your Robot off. Wait a few seconds and turn it back on again. Now try to execute your program. Remember, all data in memory, except that stored in ROM, is lost whenever power is removed.
12. Rewind your cassette recorder to where it was at the end of Step 8.
13. **RESET** your Trainer and press the **3** key to enter the Utility Mode. Then press the **4** key to enter the Trainer's upload routine. The two left-most digits of the display should read 3.4 to reflect your key entries. The other digits should remain blank. The Trainer is now awaiting input from the cassette. Note that you were not asked to enter any addresses during this operation. This is because each program is automatically replaced to its original memory locations.



14. Temporarily remove the plug in the cassette's earphone jack. Then start the cassette player. If it is properly positioned, the cassette will begin its output with a high-pitched fixed tone. Immediately following, the output will change to an alternating series of tones representing binary ones and zeros. Stop the recorder and rewind it to the beginning of the high-pitched fixed tone. Then reconnect the plug into the earphone jack.

NOTE: Due to the nature of the Robot's cassette interface, most cassette players work best when their volume control is set to maximum and their tone control is set to maximum treble.

Restart the cassette to begin uploading your program. Since placing a plug into the earphone jack mutes normal audio to the speaker, you will be unable to hear the tones as they are input to the Trainer. You can verify their presence, however, by watching for a slowly blinking decimal point between the two digits of the Trainer's display.

When all of the program's contents are entered into memory, the Trainer re-enters the Executive Mode. The actual duration of the upload process varies from seconds to minutes, depending on the length of the program.

15. Examine the contents of memory beginning at 0050, the starting address of the original program. Once you are convinced of the program's completeness, execute the program and observe the results.

## Discussion

Nonvolatile storage is extremely important in any computer environment. Without it, you, the operator, would be forced to enter and re-enter programs using time consuming step-by-step techniques, such as a keyboard. Cassette tape is just one form of nonvolatile storage; there are others. However, none of the others are so inexpensive to implement.

You may find it to your advantage to practice downloading and uploading some of your upcoming experiments. This should not only improve your proficiency in the use of cassette I/O, but will also allow you to break up your study sessions without fear of losing a keyed-in program.

## Using The Teaching Pendant

You can use the teaching pendant in either of two programming modes, the Manual Mode or the Repeat Mode. The pendant itself is a hand-held remote control box. By examining the pendant's label, you will see that, through a various combination of switches, you have total control over the Robot's seven motors. You can drive it forward and back, steer it left and right, rotate the head, raise and lower the arm, extend or retract the arm, pivot the wrist, rotate the wrist, and open and close the gripper. The vivid color-coding and descriptive symbols on the pendant's label make it an easy task to select and control motor activity.

### MANUAL PROGRAMMING MODE

When you select the Manual Mode, the Robot accepts and executes commands directly from the teaching pendant. You can drive the Robot about and control the head and arm motions. One limitation of the teaching pendant is that it can only perform a single action at any one time. You can not, for instance, drive the Robot and simultaneously turn the head. The only exception to this single motor rule involves steering. You can move and turn in the same motion.

Practice the following procedures until you are quite proficient at remote controlling your trainer.

### Procedure

16. Ensure that your Robot is fully charged. Disconnect the power cable and connect it to the teaching pendant.
17. Clear a 6ft by 6ft working space. Then place the Robot in the middle of this area.
18. Turn your Robot's power on. Initialize all the Robot's motors by using the Utility 3-1 program. Once this long initializing routine is completed, touch up initializations can be performed using the shorter 3-2 utility routine.
19. Enter the Manual Mode by pressing the 4 key. The display should have a number four and a decimal point in its furthest left LED. The other LEDs may or may not have something displayed at this time.

NOTE: Refer to Figure E18-3 and then to your teaching pendant during each of the following operations.

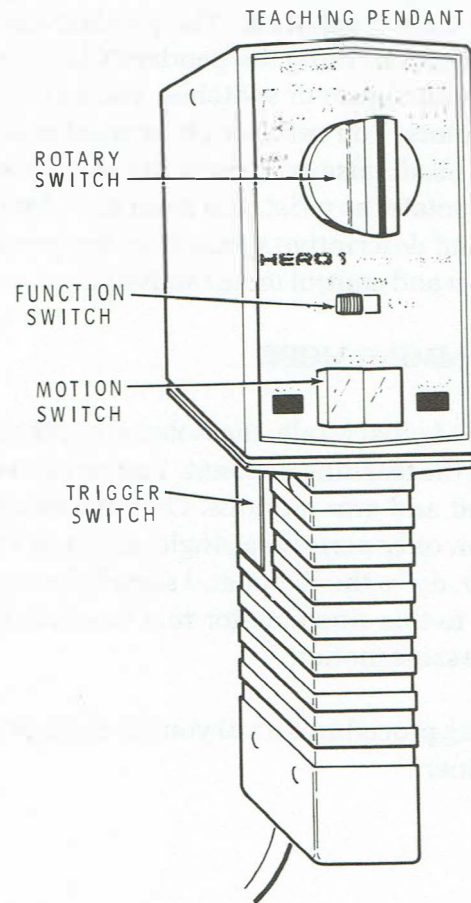


Figure E18-3

20. Turn the Rotary switch one detent to the right. This is the position used for both slow speed forward and wrist pivot up/down. Observe that nothing happens at this time.
21. If you have not already done so, move the Function switch to the **BODY** position. Note the display as you change this switch. In the arm position, zeros are displayed in the two furthest-right LEDs. In the body position, the four right-side LEDs contain data.

22. Make sure the path around the Robot is clear for at least three feet. Momentarily squeeze the Trigger switch, let the Robot move forward an inch or two, and then release the switch. Observe the changes in the four left LED displays. They indicate base movement according to pulses generated by the reflective code disk on the drive wheel. These intervals represent the same relative distance intervals you programmed into your motor control words in earlier experiments.

**Nothing operates unless you pull the Trigger. The other controls select what motors will run, their speed, and their direction, but the Trigger actually turns them on.**

23. Repeat the procedures in Step 22, letting the Robot travel a little further each time. Note the changes in the LED displays and how they automatically reset to zeros each time you stop and start. Stop at the edge of the cleared area.
24. With the Trigger released, turn the Rotary switch one detent to the left of center. This position is used both for low speed reverse and arm pivot. Hold the Teaching Pendant's cable up to keep it from getting caught by the Trainer's rear wheels, then press the Trigger. Let the Robot travel in longer and longer increments until it reaches the opposite side of the cleared area.
25. With the Trigger released, turn the Rotary switch to the second detent right of center. This position is used for both medium speed forward motion and wrist rotate. Press the Trigger and move your Trainer across the work area.
26. With the Trigger released, turn the Rotary switch to the second detent left of center. This is both the medium speed reverse drive and arm extend/retract position. Press the Trigger and move your Trainer across the work area.
27. Following the procedures outlined in the earlier steps, practice moving your Robot forward and back using its fastest speed, the third detents either side of center.
28. Place your Robot in the center of the cleared work space.

29. Place the Rotary switch in its center, Neutral, position. With your thumb, press the Motion switch to the right; then squeeze the Trigger. Observe the action of the front wheel. Also note how changes in wheel position are not reflected in the LED display.
30. While the Trigger is still depressed, release your thumb from the Motion switch and observe the front wheel as it automatically re-centers itself. Press the Motion switch alternately to the right and left and study the action of the front wheel.
31. Execute a tight right turn by first turning the wheel as in Steps 29 and 30. When the wheel is fully to the right, release the Trigger. Turn the Rotary switch to the low speed forward detent. Press the Motion switch to the right with your thumb and squeeze the Trigger. Be patient; it will take Hero a few moments to begin moving with this tight a maneuver. Release the Trigger when Hero has turned about 90 degrees.
32. Straighten the front wheel by turning the Rotary switch to neutral, releasing the Motion switch, and squeezing the Trigger.
33. Execute a tight left turn following procedures similar to those in Step 31.
34. Execute tight left and right turns in the reverse direction. Remember to turn your steering before you actually begin moving the drive.
35. Straighten your front wheel. Then place your Trainer in the center of the work area.
36. Execute standard right and left turns by allowing the drive to begin before you press the Motion switch. Practice the turning motions at slow speed. Try to make large circles, then figure-eights in both forward and reverse directions.
37. Repeat Steps 20 through 36 until you feel proficient with your manual controls.



38. Switch the Function switch to the **Arm** position. Note the LED display values as you turn the Rotary switch. Record these values below:

HEAD          ARM          ARM NEUTRAL WRIST WRIST GRIP  
ROTATE EXTEND/PIVOT PIVOT          PIVOT ROTATE

--	--	--	--	--	--	--

39. Turn the Rotary switch to the **Head** Position. Using the Motion and Trigger switches, move the head back and forth through its range of motion. Observe the LED display as the head moves. Record the LED display indications for the two extremes of Head motion.

Fully Clockwise \_\_\_\_\_ Fully Counterclockwise \_\_\_\_\_

40. Turn the Rotary switch to the **Arm Extend/Retract** position. Using both the Motion and Trigger switches, extend and retract the arm. Record these LED display indications:

Fully Retracted \_\_\_\_\_ Fully Extended \_\_\_\_\_

41. Using similar procedures to those in Steps 39 and 40. Turn the Rotary switch through the remainder of the arm and wrist functions. Practice moving each motor and record their minimum/maximum LED indications.

ARM PIVOT UP _____	ARM PIVOT DOWN _____
WRIST PIVOT UP _____	WRIST PIVOT DOWN _____
WRIST ROTATE LEFT _____	WRIST ROTATE RIGHT _____
GRIPPER OPEN _____	GRIPPER CLOSED _____

42. As in Step 38, turn the Rotary switch through its range and record the LED display values below. Compare these values with those found in Step 38.

HEAD          ARM          ARM NEUTRAL WRIST WRIST GRIP  
ROTATE EXTEND/PIVOT PIVOT          PIVOT ROTATE

--	--	--	--	--	--	--



43. Press **RESET** and use Utility Mode 3-2 to initialize the head, arm and wrist motors. Repeat Step 42.
44. Place an object, like a dixie cup, on the floor near the Robot. Pick the cup up with the gripper and move it to the Robot's other side. Use whatever motor motions necessary to perform this task. Initialize the motors and try again.

## Discussion

These first procedures were to acquaint you with base, head, arm and wrist movements under Manual control. Remember that you enter the Manual Mode from the Executive Mode by pressing the **4** key. Once in Manual, you can practice maneuvers you wish to use later in programs, or simply drive the Robot from one spot to another.

During each motion of these procedures you were asked to monitor the LED displays and, in some cases, write down what you saw. These LED displays tell you exactly how far a motor has moved (in Robot increments that is). And you can use them during program writing to determine how far a motor should move. Remember, these are the distance indications used by the Robot interpreter to control the range of all programmed motions, including the initialization procedures. A perfect example is the routine you used in earlier experiments to re-initialize the motors. The routine:

```
FD 00 00 4D 00 00 62 49
```

which you now know as the **MOVALL** instruction with its seven-byte operand, set all motors to the values indicated in the operand. These are the same values recorded in Steps 38 and 43.

Now that you are able to move your Robot around with its teaching pendant, it's time for you to learn about the Learn Mode.

## LEARN MODE

The Learn Mode is similar to the Manual Mode, with one big exception. In the Learn Mode, the Robot remembers everything you do with it. It can, therefore, repeat these motions later in the Repeat Mode. In addition, you can erase mistakes, change motor speeds, or even change the length of movements. By using the Learn Mode, you can teach the Robot to do just what you want it to (without any wasted moves) more quickly and impressively than you could do in the Manual Mode.

Before entering your final procedures for this Experiment, keep in mind the following information:

- In general, all actions are somewhat faster when played back in the Repeat Mode.
- Arm and head actions are the most reliable for Repeat Mode operations due to the inherent inaccuracies of the Robot's steering system.
- In the Learn Mode, you can insert an automatic pause between a rolling forward and a rolling backward command. A pause here improves travel accuracy. You can insert it by moving the Rotary switch through the 'N' position with the Trigger released.

## Procedure (Continued)

45. Make sure your Robot is initialized and in the Executive Mode.
46. Press the **7** key to enter the Learn Mode. The display will show a 7. with four dashes.
47. Enter the memory address where you want the Robot to begin storing the program of movements. Use 0100 for this example.
48. Enter the maximum ending address of this learning session. This is used as a preventative measure to keep you from inadvertently writing over another program higher in memory. If you exceed the storage ability determined by these addresses, the display will show "FULL!" and the Robot will refuse further commands.

NOTE: After you enter the second memory address, the display will show 7.F.0108. This is the first available address open to receive your input. A MVAL instruction is automatically inserted into the first eight bytes of every learn program. This insures that the Robot will return to its original position when the program is played back.

49. Use the teaching pendant to do the following:
  - A. Rotate the head so the arm is on its left side.
  - B. Raise the arm to the horizontal position.
  - C. Drop the wrist in line with the arm.
  - D. Open the gripper.
  - E. Extend the arm about 2/3 range.
  - F. Move forward 12 inches at low speed.
50. When you are done with all movements, press **RESET** to end the learning session.
51. Play back your program by pressing the **A** key (Repeat Mode), followed by the **DO** key and the program's starting address, which was 0100 in this first example. Observe your Robot as it replays what it learned from you. Make sure that you start from the same position as when you entered the program.
52. Use the EXAMine Mode to view and record the program you entered during this learning session. Then compare yours with the one in Figure E18-4.

ADDRESS	HEX DATA	OPCODE OPERAND	COMMENTS
0100	FD 00 00 4d 00 00 62 49	MVAL 00 00 4D 00 00 62 49	Initialize motors.
0108	D3 C8 37	MVRELW C8 37	Move head relative right 37.
010B	D3 48 51	MVRELW 48 51	Move arm relative up 51.
010E	D3 88 55	MVRELW 88 55	Move wrist pivot down 55.
0111	D3 A8 70	MVRELW A8 70	Move gripper open 70.
0114	D3 28 4A	MVRELW 28 48	Move arm extend 48.
0117	8F 00 20	PAUSE 00 20	Pause for 2 seconds.
011A	C3 08 16	MVWI 08 16	Move extended and wait — base ahead 16.
011D	3A		

Figure E18-4  
Program from Learn Mode.

## Discussion

Each time you pulled the Trigger on your teaching pendant, another instruction is stored in memory; telling which motor was selected, which direction it traveled and how far it went. In some cases, multiple instructions are stored for just one of your moves. Examples would include turning and moving over large distances on only one Trigger squeeze. You may have noticed that you wasted some memory by going too far or not far enough and then having to compensate with additional moves. Duplicated motions use up twice as much memory. One way to minimize this memory waste is to practice your routines and then practice them some more, and although practice is always a good idea, there is also a special feature built into the interpreter that gives you a little editing power too.

### THE BACK-UP FEATURE

During the Learn Mode, your Robot has the ability to 'back-up' both its arm and head motions and its memory. If you enter a wrong motion, you can back up and re-do that step.

You begin backing up arm and head motions by pressing the **B** key. Then, when you depress the Trigger, the Robot begins to back through the program. You can back part way through a step, or all the way. At the end (true beginning) of each back-up step, the Robot will flash a "b" on the display. This prompt warns you that you should release the Trigger; otherwise, the Robot will begin backing through the next step. If you continue to hold the Trigger, the back-up function will take you all the way to the start of the program. As you back through, you will see the memory address display and motor position display count backwards.

You may stop backing up at any time, midway through a step or between steps. Nothing is actually changed in memory until you press **FWD** (forward) and enter a new step.

If you just used a back-up function to retrace your movements and don't want to change anything, just press **RESET**.

## THE REVERSE INSTRUCTION FEATURE

Another interesting feature available in the Learn Mode is the Reverse Instruction Feature (RTI). If you are doing an operation that requires the Robot to do something, and then backup and undo the same thing, you can use the RTI feature to get the interpreter to generate an exact reverse of one or more connected instructions. The RTI feature, like back-up, only works with the head and arm motors.

If you have just finished a step that you would like to reverse, press the **RTI** key (7). An "r" for reverse will show on the display. When you pull the Pendant's Trigger, the arm or head will begin to back through the last instruction. The operation proceeds just like the Back-up feature, one step at a time with a flashing "r" and a pause between steps. However, each step you reverse through is actually added to memory. To stop the RTI feature, simply press the **FWD** key and begin entering new steps with the Teaching Pendant, or press **RESET**.

## Procedure (Continued)

52. Initialize your Robot's head and arm motors.
53. Enter the Learn Mode and do the following:
  - A. Move the head counterclockwise about 90 degrees.
  - B. Move the head clockwise about 45 degrees.
  - C. Press the Back-up key (**B**).
  - D. Squeeze the Trigger on the Teaching Pendant. Hold down until a flashing "b" appears on the displays. You have now backed through the 45 degree clockwise rotation.
  - E. Resume teaching by pressing the **FWD** key.
  - F. Move the head counterclockwise another 45 degrees.
  - G. Press the **RESET** key.
54. Use the Repeat Mode to play back your edited program. Notice that the head comes back to home automatically before it begins executing your first 90 degree rotation.

This initialization is generated by the MVAL instruction automatically placed in the first eight bytes of every learned program.

Once the head found home, it performed one noticable counterclockwise motion. This one motion was the sum of the two counterclockwise motions that you entered.



55. Initialize your Robot's head and arm. Enter the Learn Mode and do the following:
- A. Raise the arm about 20 degrees.
  - B. Extend the arm to within 1 inch of the floor.
  - C. Lower the wrist pivot so the wrist is parallel to the floor.
  - D. Open the gripper fully.
  - E. Rotate the head clockwise 90 degrees.
  - F. Close the gripper.
  - G. Press the **RTI** key (7).
  - H. Squeeze the Trigger and hold it while the interpreter retraces through each of your earlier motor movements. Note the blinking "r" between retrace steps.
  - I. Press **RESET**.
56. Repeat your program using the Repeat Mode. Observe how the RTI function duplicated your earlier instructions only in reverse.

## Discussion

Now you know how to use both your Trainer's Cassette I/O utilities and its Manual and Learn Modes. To get truly proficient with these features, you will need more practice than what we could offer here. Develop some practical experiments of your own to test your skills. Remember the **ABORT** button on top, should something not go exactly as you planned.



1. The first step in the process of creating a new product is to identify a market need.

2. The next step is to develop a concept for the product.
3. This is followed by a detailed design of the product.
4. The design is then used to create a prototype.
5. The prototype is used to test the product.
6. The results of the testing are used to refine the design.
7. The refined design is then used to create the final product.
8. The final product is then marketed to the target market.
9. The marketing process involves identifying the target market, developing a marketing strategy, and implementing the strategy.
10. The final step in the process is to evaluate the success of the product.

11. The success of a product is determined by its ability to meet the needs of the target market.

## Unit 12: The Product

12. The product is the result of the design and development process.

13. The product is the result of the design and development process.

14. The product is the result of the design and development process.

15. The product is the result of the design and development process.

16. The product is the result of the design and development process.

17. The product is the result of the design and development process.

18. The product is the result of the design and development process.

19. The product is the result of the design and development process.

20. The product is the result of the design and development process.

## Experiment 19

### *Modifying A Taught Program*

**OBJECTIVES:**

*To program the Robot Trainer to perform a repetitive task using the “lead-through” method of teaching.*

*To illustrate how a taught program can be modified to include pauses, delete unwanted motions, and make the program repetitive.*

## **Introduction**

In Unit 11 you learned the two basic methods — “walk-through” and “lead-through” teaching — by which industrial robots are taught to perform a routine. You also learned that once a robot has been taught a specific routine, you can modify the routine by changing portions of the program with keyboard entries.

In this experiment, you will teach the Robot a simple routine using the hand-held teaching pendant. Once the routine has been taught, you will, through keyboard entries, change the program so it will repeat the routine as long as you desire. You will also see how a pause or wait command is inserted into the program.

## **Material Required**

ET-18 Robot Trainer

## Procedure

1. Place the Robot Trainer in the center of an open area approximately 6 x 6 feet. Connect the teaching pendant to the Trainer. Energize and initialize the Trainer.
2. Press the **7** key to enter the Learn Mode. The display will show a 7, followed by four dashes. Enter memory address 0100. The display will again show a 7, followed by four dashes. Enter memory address 0150. After you enter the second memory address, the display will show 7F0108.
3. In this step, make sure that you **DO NOT** pull the Trigger switch while setting the other switches. On the teaching pendant, put the Rotary switch in the **HEAD** position and the Function switch in the **ARM** position.
4. As you pull the Trigger also depress and hold the **MOTION** switch to the **LEFT** position. The head will move counterclockwise. Allow it to move until it reaches its limit of travel (where it will automatically stop). Release the Trigger and Motion switches.
5. Record the data shown on the display \_\_\_\_\_. The data should be 7F010b.
6. Pull the Trigger, but this time depress and hold the Motion switch in the **RIGHT** position. The head will move clockwise. Allow it to travel to its approximate starting position, straight ahead. Release the Trigger and Motion switches.
7. Record the data shown on the display \_\_\_\_\_. The data should be 7F010E.
8. Press the **RESET** key.
9. "Home" the Robot's head by pressing the **3** key and then the **2** key.
10. Have the Robot execute its learned program by pressing the **A** and **D** keys, followed by starting memory address 0100. Then read the following discussion.

## Discussion

In the previous steps, you used the teaching pendant to teach the Robot a very simple task. This is very similar to the way a large, more sophisticated industrial robot is taught. Using the method shown above, you could just as easily teach the Trainer to drive any one of its manipulator axes to a specific position.

In Step 2, you entered the first memory address (0100). This is where the Robot began storing the program movements. The second memory address you entered (0150) was the last address the Robot could use for this program. This “stopping” address prevents you from running into any other program you may already have in memory. If for some reason you use all the memory allotted for yourself, the display will show “FULL!” This feature is especially useful when you are teaching very long programs, requiring large amounts of memory.

When you executed the learned program, you should have noticed that the Robot moved its head to the counterclockwise limit and then returned to the approximate starting position. This was accomplished with little or no “pause” in the movements. In the following procedure, you will, through keyboard entries, change the distance of travel, make the program repeat, and add a short pause to the routine.

## Procedure (Continued)

11. Examine and record the data stored in the learned routine.

<u>MEMORY ADDRESS</u>	<u>DATA</u>
0100	_____
0101	_____
0102	_____
0103	_____
0104	_____
0105	_____
0106	_____
0107	_____
0108	_____
0109	_____
010A	_____
010B	_____
010C	_____
010D	_____
010E	_____

12. Change the contents of memory address 010A and 010D to reflect a decrease of 30 to the data stored there. For example, if the number at memory address location 010A is 62, change it to read 32. In the same manner, if the number at memory address location 010D is 5F, change it to read 2F. Reset the Trainer.



13. "Home" the Trainer by pressing the **3** and **2** keys and then execute the program. Observe the movement of the Trainer's turret. Did the amount of travel increase or decrease? \_\_\_\_\_.
14. In the same manner change the contents of memory address 010E to 20, and the contents of memory address 010F to F0. Reset the Trainer. Now "home" the turret and execute the program. Observe the movement of the Trainer's turret. Does the learned routine repeat? \_\_\_\_\_
15. Stop the Trainer's movements by pressing the **RESET** key. Again, home the Trainer.
16. Modify the program to reflect the following changes:

<u>Memory Address</u>	<u>Change To</u>
010E	8F
010F	00
0110	40
0111	20
0112	ED

Reset the Trainer.

17. Execute the program. Does the Trainer perform the taught routine and then pause approximately 4 seconds before repeating the routine? \_\_\_\_\_

18. **RESET** and home the trainer. Examine the program and record the data below.

MEMORY ADDRESS	DATA
-------------------	------

0100	_____
------	-------

0101	_____
------	-------

0102	_____
------	-------

0103	_____
------	-------

0104	_____
------	-------

0105	_____
------	-------

0106	_____
------	-------

0107	_____
------	-------

0108	_____
------	-------

0109	_____
------	-------

010A	_____
------	-------

010B	_____
------	-------

010C	_____
------	-------

010D	_____
------	-------

010E	_____
------	-------

010F	_____
------	-------

0110	_____
------	-------

0111	_____
------	-------

0112	_____
------	-------

19. Read the following discussion.

## Discussion

In this portion of the experiment, you changed the previously taught program by decreasing the amount of turret movement, making the routine repetitive, and inserting a pause between the repetitive movements. These modifications were inserted through keyboard entries in much the same manner as with industrial robots. When you examined and recorded the data in Step 11, your program should have been similar to the one shown below.

<u>MEMORY ADDRESS</u>	<u>DATA</u>	<u>COMMENTS</u>
0100	FD	The first seven bytes of data in the program is a preprogrammed function used to "set-up" a robot routine. We are not concerned with this data. The only data we are concerned with starts at memory address 0108.
0101	00	
0102	00	
0103	4D	
0104	00	
0105	00	
0106	62	
0107	49	This is robot code to drive a motor.
0108	D3	
0109	CC	This is an instruction telling the turret motor to drive counterclockwise at a given speed.
010A	XX	This number is an instruction telling the turret motor how far to drive CCW.
010B	D3	This is robot code to drive a motor.
010C	C8	This is an instruction telling the turret motor to drive clockwise at a given speed.
010D	XX	This number is an instruction telling the turret motor how far to drive CW.
010E	3A	This informed the MPU that the program is complete, and it should return to the Executive Mode.

The distance the turret traveled was modified in Step 12. Recall that you decreased the distance of travel by decreasing the magnitude of the number at memory locations 010A and 010D. These numbers could have been decreased more or less, depending on how much you wanted to decrease the amount of turret travel. You also could have increased the amount of travel by making these numbers larger. However, since you were already in the counterclockwise limit, you could not increase the amount of travel in that direction.

You modified the taught routine even more in Step 14. Here, you changed the contents of the program so the turret would repeat the routine. This was accomplished by removing the 3A command, from memory address 010E, which told the Trainer to go back to the Executive Mode when it finished the routine. In its place, you added 20, relative branch always instruction. Since you had to tell the MPU where to branch to, you had to add another instruction at 010F, which was F0. This offset instruction told the MPU to count back (designated by the F portion of the instruction F0) from where it was currently located, in this case 010F.

The 0 portion of the F0 instruction tells the MPU how far to count back, starting from 010F. Thus, you count backwards from 010F to 010E then to 010D....until you reach the point in the program where you want to start the repeat, which in this case is 0100, the start of the program. We went back to the beginning of the program because we wanted to repeat the complete program. If you desired to repeat just a portion of the program, you would only count back as far as necessary to pick up the beginning of that portion.

The final modification to your program was the pause you put in by entering the 3-byte instruction 8F 00 40 just before you branched back to repeat the routine. 8F is the Trainer's opcode for initiating a pause. The next 2-bytes of the instruction, in this case 00 and 40, are hexadecimal numbers telling the MPU how long the pause should last. Each segment is about 1/16 of a second. Thus, "8F0040" would mean a pause of 64 segments (remember 40 hex is 64), or about 4 seconds. You could add a pause lasting from a portion of a second to 4096 seconds in duration. In addition, this pause could be added at any memory address location in the learned program.

As you can see by the above discussion, a taught program can easily be modified. This is especially valuable when you are teaching an industrial robot a specific routine. For instance, if you made an error in the amount of travel when you were teaching the Robot a routine, you could easily correct the error. And once you had the routine exactly as you wanted it to perform a given task, you could easily make it repetitive. If for some reason the Robot had to wait before repeating the task, for example waiting for a workpiece to arrive or a machine to finish a process, a pause could be inserted in that portion of the taught program.

The data you recorded in Step 18 is to help you become familiar with programming modifications. We suggest that you modify several of the commands to change the amount of travel and the length of the pauses.



## Appendix A

# NUMBER SYSTEMS AND CODES



## DECIMAL NUMBER SYSTEM

A basic distinguishing feature of a number system is its **base** or **radix**. The base indicates the number of characters or digits used to represent quantities in that number system. The decimal number system has a base or radix of 10 because we use the ten digits 0 through 9 to represent quantities. When a number system is used where the base is not known, a subscript is used to show the base. For example, the number  $4603_{10}$  is derived from a number system with a base of 10.

**Positional Notation** The decimal number system is positional or weighted. This means each digit position in a number carries a particular weight which determines the magnitude of that number. Each position has a weight determined by some power of the number system base, in this case 10. The positional weights are  $10^0$  (units)\*,  $10^1$  (tens),  $10^2$  (hundreds), etc. Refer to Figure 1-1 for a condensed listing of powers of 10.

$10^0$	= 1
$10^1$	= 10
$10^2$	= 100
$10^3$	= 1,000
$10^4$	= 10,000
$10^5$	= 100,000
$10^6$	= 1,000,000
$10^7$	= 10,000,000
$10^8$	= 100,000,000
$10^9$	= 1,000,000,000

Figure 1-1  
Condensed listing of powers of 10.

\*Any number with an exponent of zero is equal to one.

We evaluate the total quantity of a number by considering the specific digits and the weights of their positions. For example, the decimal number 4603 is written in the shorthand notation with which we are all familiar. This number can also be expressed with positional notation.

$$\begin{aligned}(4 \times 10^3) + (6 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) &= \\(4 \times 1000) + (6 \times 100) + (0 \times 10) + (3 \times 1) &= \\4000 + 600 + 0 + 3 &= 4603_{10}\end{aligned}$$

To determine the value of a number, multiply each digit by the weight of its position and add the results.

**Fractional Numbers** So far, only **integer** or whole numbers have been discussed. An integer is any of the natural numbers, the negatives of these numbers, or zero (that is, 0, 1, 4, 7, etc.). Thus, an integer represents a whole or complete number. But, it is often necessary to express quantities in terms of fractional parts of a whole number.

Decimal fractions are numbers whose positions have weights that are **negative powers of ten** such as  $10^{-1} = \frac{1}{10} = 0.1$ ,  $10^{-2} = \frac{1}{100} = 0.01$ , etc.

Figure 1-2 provides a condensed listing of negative powers of 10 (decimal fractions).

$$10^{-1} = \frac{1}{10} = 0.1$$

$$10^{-2} = \frac{1}{100} = 0.01$$

$$10^{-3} = \frac{1}{1000} = 0.001$$

$$10^{-4} = \frac{1}{10,000} = 0.0001$$

$$10^{-5} = \frac{1}{100,000} = 0.00001$$

$$10^{-6} = \frac{1}{1,000,000} = 0.000001$$

Figure 1-2  
Condensed listing of negative  
powers of 10.



A radix point (decimal point for base 10 numbers) **separates** the **integer** and **fractional** parts of a number. The integer or whole portion is to the left of the decimal point and has positional weights of units, tens, hundreds, etc. The fractional part of the number is to the right of the decimal point and has positional weights of tenths, hundredths, thousandths, etc. To illustrate this, the decimal number 278.94 can be written with positional notation as shown below.

$$\begin{aligned}(2 \times 10^2) + (7 \times 10^1) + (8 \times 10^0) + (9 \times 10^{-1}) + (4 \times 10^{-2}) &= \\(2 \times 100) + (7 \times 10) + (8 \times 1) + (9 \times 1/10) + (4 \times 1/100) &= \\200 + 70 + 8 + 0.9 + 0.04 &= 278.94_{10}\end{aligned}$$

In this example, the left-most digit ( $2 \times 10^2$ ) is the **most significant digit** or MSD because it carries the greatest weight in determining the value of the number. The right-most digit, called the **least significant digit** or LSD, has the lowest weight in determining the value of the number.

## BINARY NUMBER SYSTEM

The simplest number system that uses positional notation is the binary number system. As the name implies, a **binary** system contains only two elements or states. In a number system this is expressed as a base of 2, using the digits 0 and 1. These two digits have the same basic value as 0 and 1 in the decimal number system.

### Positional Notation

As with the decimal number system, each bit (digit) position of a binary number carries a particular weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example 2). To evaluate the total quantity of a number, consider the specific bits and the weights of their positions. (Refer to Figure 1-3 for a condensed listing of powers of 2.) For example, the binary number 110101 can be written with positional notation as follows:

$$(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

To determine the decimal value of the binary number 110101, multiply each bit by its positional weight and add the results.

$$(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 32 + 16 + 0 + 4 + 0 + 1 = 53_{10}$$

$2^0 = 1_{10}$	$2^6 = 64_{10}$
$2^1 = 2_{10}$	$2^7 = 128_{10}$
$2^2 = 4_{10}$	$2^8 = 256_{10}$
$2^3 = 8_{10}$	$2^9 = 512_{10}$
$2^4 = 16_{10}$	$2^{10} = 1024_{10}$
$2^5 = 32_{10}$	$2^{11} = 2048_{10}$

Figure 1-3  
Condensed listing of powers of 2.



Fractional binary numbers are expressed as negative powers of 2. Figure 1-4 provides a condensed listing of negative powers of 2. In positional notation, the binary number 0.1101 can be expressed as follows:

$$(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

To determine the decimal value of the binary number 0.1101, multiply each bit by its positional weight and add the results.

$$(1 \times 1/2) + (1 \times 1/4) + (0 \times 1/8) + (1 \times 1/16) = \\ 0.5 + 0.25 + 0 + 0.0625 = 0.8125_{10}$$

In the binary number system, the radix point is called the binary point.

$$2^{-1} = \frac{1}{2} = 0.5_{10}$$

$$2^{-2} = \frac{1}{4} = 0.25_{10}$$

$$2^{-3} = \frac{1}{8} = 0.125_{10}$$

$$2^{-4} = \frac{1}{16} = 0.0625_{10}$$

$$2^{-5} = \frac{1}{32} = 0.03125_{10}$$

$$2^{-6} = \frac{1}{64} = 0.015625_{10}$$

$$2^{-7} = \frac{1}{128} = 0.0078125_{10}$$


$$2^{-8} = \frac{1}{256} = 0.00390625_{10}$$

Figure 1-4  
Condensed listing of negative  
powers of 2.

## Converting Between the Binary and Decimal Number Systems

**Binary to Decimal** To convert a binary number into its decimal equivalent, add together the weights of the positions in the number where binary 1's occur. The weights of the integer and fractional positions are indicated below.

INTEGER								FRACTIONAL		
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$
128	64	32	16	8	4	2	1	.5	.25	.125

Binary Point 

As an example, convert the binary number 1010 into its decimal equivalent. The right-most bit, called the **least significant bit** or LSB, has the lowest integer weight of  $2^0 = 1$ . The left-most bit is the **most significant bit** (MSB) because it carries the greatest weight in determining the value of the number. In this example, it has a weight of  $2^3 = 8$ . To evaluate the number, add together the weights of the positions where binary 1's appear. In this example, 1's occur in the  $2^3$  and  $2^1$  positions. The decimal equivalent is ten.

Binary Number	1	0	1	0					
Position Weights	$2^3$	$2^2$	$2^1$	$2^0$					
Decimal Equivalent	8	+	0	+	2	+	0	=	$10_{10}$

**Decimal to Binary** A decimal integer number can be converted to a different base or radix through successive divisions by the desired base. To convert a decimal integer number to its binary equivalent, successively divide the number by 2 and note the remainders. When you divide by 2, the remainder will always be 1 or 0.

The remainders form the equivalent binary number.



As an example, the decimal number 25 is converted into its binary equivalent.

$$\begin{array}{rcl}
 25 \div 2 = 12 & \text{with remainder } 1 & \leftarrow \text{LSB} \\
 12 \div 2 = 6 & & 0 \\
 6 \div 2 = 3 & & 0 \\
 3 \div 2 = 1 & & 1 \\
 1 \div 2 = 0 & & 1 \leftarrow \text{MSB}
 \end{array}$$

Divide the decimal number by 2 and note the remainder. Then divide the quotient by 2 and again note the remainder. Then divide the quotient by 2 and again note the remainder. Continue this division process until 0 results. Then collect remainders beginning with the last or most significant bit (MSB) and proceed to the first or least significant bit (LSB). The number  $11001_2 = 25_{10}$ . Notice that the remainders are collected in the reverse order. That is, the first remainder becomes the least significant bit, while the last remainder becomes the most significant bit.

**NOTE:** Do not attempt to use a calculator to perform this conversion. It would only supply you with confusing results.

To convert a decimal fraction to a different base or radix, multiply the fraction successively by the desired base and record any integers produced by the multiplication as an overflow. For example, to convert the decimal fraction 0.3125 into its binary equivalent, multiply repeatedly by 2.

$$\begin{array}{rcl}
 0.3125 \times 2 = 0.625 = 0.625 & \text{with overflow } 0 & \leftarrow \text{MSB} \\
 0.6250 \times 2 = 1.250 = 0.250 & & 1 \\
 0.2500 \times 2 = 0.500 = 0.500 & & 0 \\
 0.5000 \times 2 = 1.000 = 0 & & 1 \leftarrow \text{LSB}
 \end{array}$$

These multiplications will result in numbers with a 1 or 0 in the units position (the position to the left of the decimal point). By recording the value of the units position, you can construct the equivalent binary fraction. This units position value is called the "overflow." Therefore, when 0.3125 is multiplied by 2, the overflow is 0. This becomes the most significant bit (MSB) of the binary equivalent fraction. Then 0.625 is multiplied by 2. Since the product is 1.25, the overflow is 1. When there is an overflow of 1, it is effectively subtracted from the product when the value is recorded. Therefore, only 0.25 is multiplied by 2 in the next multiplication process. This method continues until an overflow with no fraction results. It is important to note that you can not always obtain 0 when you multiply by 2. Therefore, you should only continue the conver-

sion process to the precision you desire. Collect the conversion overflows beginning at the radix (binary) point with the MSB and proceed to the LSB. This is the same order in which the overflows were produced. The number  $0.0101_2 = 0.3125_{10}$ .

If the decimal number contains both an integer and fraction, you must separate the integer and fraction using the decimal point as the break point. Then perform the appropriate conversion process on each number portion. After you convert the binary integer and binary fraction, recombine them. For example, the decimal number 14.375 is converted into its binary equivalent.

$$14.375_{10} = 14_{10} + 0.375_{10}$$

$$14 \div 2 = 7$$

$$7 \div 2 = 3$$

$$3 \div 2 = 1$$

$$1 \div 2 = 0$$

with remainder 0 ← LSB

1

1

1 ← MSB

$$\boxed{14_{10} = 1110_2}$$

$$0.375 \times 2 = 0.75 = 0.75$$

$$0.750 \times 2 = 1.50 = 0.50$$

$$0.500 \times 2 = 1.00 = 0$$

with overflow

0 ← MSB

1

1 ← LSB

$$\boxed{0.375_{10} = 0.011_2}$$

$$14.375_{10} = 14_{10} + 0.375_{10} = 1110_2 + 0.011_2 = 1110.011_2.$$



## HEXADECIMAL NUMBER SYSTEM

Hexadecimal is another number system that is often used with microprocessors. As the name implies, hexadecimal has a base (radix) of  $16_{10}$ . It uses the digits 0 through 9 and the letters A through F.

The letters are used because it is necessary to represent  $16_{10}$  different values with a single digit for each value. Therefore, the letters A through F are used to represent the number values  $10_{10}$  through  $15_{10}$ . The following discussion will compare the decimal number system with the hexadecimal number system.

All of the numbers are of equal value between systems ( $0_{10} = 0_{16}$ ,  $3_{10} = 3_{16}$ ,  $9_{10} = 9_{16}$ , etc.). For numbers greater than 9, this relationship exists:  $10_{10} = A_{16}$ ,  $11_{10} = B_{16}$ ,  $12_{10} = C_{16}$ ,  $13_{10} = D_{16}$ ,  $14_{10} = E_{16}$ , and  $15_{10} = F_{16}$ . Using letters in counting may appear awkward until you become familiar with the system. Figure 1-8 illustrates the relationship between decimal, hexadecimal, and binary integers, while Figure 1-9 illustrates the relationship between decimal, hexadecimal, and binary fractions.

DECIMAL	HEXADECIMAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

Figure 1-8

Sample comparison of decimal,  
hexadecimal, and binary integers.

DECIMAL	HEXADECIMAL	BINARY
0.00390625	0.01	0.00000001
0.0078125	0.02	0.0000001
0.01171875	0.03	0.00000011
0.015625	0.04	0.000001
0.01953125	0.05	0.00000101
0.0234375	0.06	0.0000011
0.02734375	0.07	0.00000111
0.03125	0.08	0.00001
0.03515625	0.09	0.00001001
0.0390625	0.0A	0.0000101
0.04296875	0.0B	0.00001011
0.046875	0.0C	0.00011
0.05078125	0.0D	0.00001101
0.0546875	0.0E	0.0000111
0.05859375	0.0F	0.00001111

Figure 1-9

Sample comparison of decimal,  
hexadecimal, and binary fractions.



As with the previous number systems, each digit position of a hexadecimal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example,  $16_{10}$ ). The total quantity of a number can be evaluated by considering the specific digits and the weights of their positions. (Refer to Figure 1-10 for a condensed listing of powers of  $16_{10}$ .) For example, the hexadecimal number E5D7.A3 can be written with positional notation as follows:

$$(E \times 16^3) + (5 \times 16^2) + (D \times 16^1) + (7 \times 16^0) + (A \times 16^{-1}) + (3 \times 16^{-2})$$

The decimal value of the hexadecimal number E5D7.A3 is determined by multiplying each digit by its positional weight and adding the results. As with the previous number systems, the radix (hexadecimal) point separates the integer from the fractional part of the number.

$$(14 \times 4096) + (5 \times 256) + (13 \times 16) + (7 \times 1) + (10 \times 1/16) + (3 \times 1/256) = 57344 + 1280 + 208 + 7 + 0.625 + 0.01171875 = 58839.63671875_{10}$$

$16^{-4}$	$=$	$\frac{1}{65536}$	$=$	$0.0000152587890625_{10}$
$16^{-3}$	$=$	$\frac{1}{4096}$	$=$	$0.000244140625_{10}$
$16^{-2}$	$=$	$\frac{1}{256}$	$=$	$0.00390625_{10}$
$16^{-1}$	$=$	$\frac{1}{16}$	$=$	$0.0625_{10}$
$1_{10}$	$=$	$16^0$		
$16_{10}$	$=$	$16^1$		
$256_{10}$	$=$	$16^2$		
$4096_{10}$	$=$	$16^3$		
$65536_{10}$	$=$	$16^4$		
$1048576_{10}$	$=$	$16^5$		
$16777216_{10}$	$=$	$16^6$		

Figure 1-10  
Condensed listing of powers of 16.

## Conversion From Decimal to Hexadecimal

Decimal to hexadecimal conversion is accomplished in the same manner as decimal to binary, but with a base number of  $16_{10}$ . As an example, the decimal number 156 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 156 \div 16 = 9 & \text{with remainder } 12 = C & \leftarrow & \text{LSD} \\ 9 \div 16 = 0 & & 9 = 9 & \leftarrow \text{MSD} \end{array}$$

Divide the decimal number by  $16_{10}$  and note the remainder. If the remainder exceeds 9, convert the 2-digit number to its hexadecimal equivalent ( $12_{10} = C$  in this example). Then divide the quotient by 16 and again note the remainder. Continue dividing until a quotient of 0 results. Then collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number  $9C_{16} = 156_{10}$ . NOTE: The letter H after a number is sometimes used to indicate hexadecimal.

To convert a decimal fraction to a hexadecimal fraction, multiply the fraction successively by  $16_{10}$  (hexadecimal base). As an example the decimal fraction 0.78125 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 0.78125 \times 16 = 12.5 = 0.5 & \text{with overflow} & 12 = C & \leftarrow \text{MSD} \\ 0.50000 \times 16 = 8.0 = 0 & & 8 = 8 & \leftarrow \text{LSD} \end{array}$$

Multiply the decimal by  $16_{10}$ . If the product exceeds one, subtract the integer (overflow) from the product. If the “overflow” exceeds 9, convert the 2-digit number to its hexadecimal equivalent. Then multiply the product fraction by  $16_{10}$  and again note any overflow. Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 16. Therefore, you should only continue the conversion to the precision you desire. Collect the conversion overflows beginning at the radix point with the MSD and proceed to the LSD. The number  $0.C8_{16} = 0.78125_{10}$ .



As shown in this section, conversion of an integer from decimal to hexadecimal requires a different technique than for conversion of a fraction. Therefore, when you convert a hexadecimal number composed of an integer and a fraction, you must separate the integer and fraction, then perform the appropriate operation on each. After you convert them, you must recombine the integer and fraction. For example, the decimal number 124.78125 is converted into its hexadecimal equivalent.

$$124.78125_{10} = 124_{10} + 0.78125_{10}$$

$$124 \div 16 = 7 \quad \text{with remainder } 12 = C \quad \leftarrow \text{LSD}$$

$$7 \div 16 = 0 \quad \quad \quad 7 = 7 \quad \leftarrow \text{MSD}$$

$$124_{10} = 7C_{16}$$

$$0.78125 \times 16 = 12.5 = 0.5 \quad \text{overflow} \quad 12 = C \quad \leftarrow \text{MSD}$$

$$0.50000 \times 16 = 8.0 = 0 \quad \quad \quad 8 = 8 \quad \leftarrow \text{LSD}$$

$$0.78125_{10} = 0.C8_{16}$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 7C_{16} + 0.C8_{16} = 7C.C8_{16}$$

First separate the decimal integer and fraction. Then convert the integer and fraction to hexadecimal.

Finally, recombine the integer and fraction.

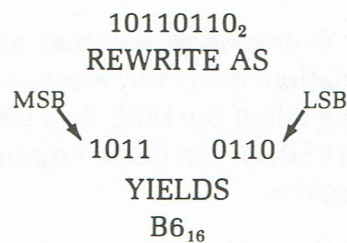
## Converting Between the Hexadecimal and Binary Number Systems

The hexadecimal number system is an excellent shorthand form to express large binary quantities. Figures 1-8 and 1-9 illustrate the relationship between hexadecimal and binary integers and fractions.

As you know, four bits of a binary number exactly equal  $16_{10}$  value combinations. Therefore, you can represent a 4-bit binary number with a 1-digit hexadecimal number:

$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13_{10} = D_{16}$$

Because of this relationship, converting binary to hexadecimal is simple and straightforward. For example, binary number 10110110 is converted into its hexadecimal equivalent.



To convert a binary number to hexadecimal, first separate the number into groups containing four bits, beginning with the least significant bit. Then convert each 4-bit group into its hexadecimal equivalent. Don't forget to use letter digits as required. This gives you a hexadecimal number equal in value to the binary number.



Binary fractions can also be converted to their hexadecimal equivalents using the same process, with one exception; the binary bits are separated into groups of four, beginning with the most significant bit (at the radix point). For example, the binary fraction 0.01011011 is converted into its hexadecimal equivalent.

0.01011011<sub>2</sub>  
REWRITE AS  
MSB                      LSB  
    ↙                    ↘  
0.0101                  1011  
YIELDS  
0.5B<sub>16</sub>

Again, you must separate the binary number into groups of four, beginning with the radix point. Then convert each 4-bit group into its hexadecimal equivalent. This gives you a hexadecimal number equal in value to the binary number.

As with octal to binary conversions, you may add zeros to fill out the binary number for calculation. However, when you add zeros to a binary integer, place them to the left of the MSB. In a binary fraction, zeros are placed to the right of the LSB. Never, under any circumstances, move the radix to perform conversions.

Now, a binary number containing both an integer and a fraction ( $110110101.01110111_2$ ) will be converted into its hexadecimal equivalent.

$110110101.01110111_2$   
 REWRITE AS

$\begin{array}{ccccccc} \text{MSB} \swarrow & & & & & & \nwarrow \text{LSB} \\ \underline{0001} & 1011 & 0101.0111 & 0111 \end{array}$

YIELDS  
 $1B5.77_{16}$

The integer part of the number is separated into groups of four, **beginning** at the radix point. Note that three zeros were added to the third group to complete the group. The fractional part of the number is separated into groups of four, **beginning** at the radix point. (No zeros were needed to complete the fractional groups.) The integer and fractional 4-bit groups are then converted to hexadecimal. The number  $110110101.01110111_2 = 1B5.77_{16}$ . **Never** shift the radix point in order to form 4-bit groups.

Converting hexadecimal to binary is just the opposite of the previous process; simply convert each hexadecimal number into its 4-bit binary equivalent. For example, convert the hexadecimal number  $8F.41_{16}$  into its binary equivalent.

$8F.41_{16}$   
 YIELDS

$\begin{array}{ccccccc} \text{MSB} \swarrow & & & & & & \nwarrow \text{LSB} \\ 1000 & 1111.0100 & 0001 \end{array}$

REWRITE AS  
 $10001111.01000001_2$

Convert each hexadecimal digit into a 4-bit binary number. Then condense the 4-bit groups to form the binary value equal to the hexadecimal value. The number  $8F.41_{16} = 10001111.01000001_2$ .



## BINARY CODES

Converting a decimal number into its binary equivalent is called "coding." A decimal number is expressed as a binary code or binary number. The **binary number system**, as discussed, is known as the pure binary code. This name distinguishes it from other types of binary codes. This section will discuss some of the other types of binary codes used in computers.

### Binary Coded Decimal

It is difficult to quickly glance at a binary number and recognize its decimal equivalent. For example, the binary number 1010011 represents the decimal number 83. However, within a few minutes, using the procedures described earlier, you could readily calculate its decimal value. The amount of time it takes to convert or recognize a binary number quantity is a distinct disadvantage in working with this code despite the numerous hardware advantages. Engineers recognized this problem early and developed a special form of binary code that was more compatible with the decimal system. This special compromise code is known as binary coded decimal (BCD). The BCD code combines some of the characteristics of both the binary and decimal number systems.



**8421 BCD Code** The BCD code is a system of representing the decimal digits 0 through 9 with a 4-bit binary code. This BCD code uses the standard 8421 position **weighting system** of the pure binary code. The standard 8421 BCD code and the decimal equivalents and binary are shown in Figure 1-11. As with the pure binary code, you can convert the BCD numbers into their decimal equivalents by simply adding together the weights of the bit positions whereby the binary 1's occur. Note, however, that there are only ten possible valid 4-bit code arrangements. The 4-bit binary numbers representing the decimal numbers 10 through 15 are invalid in the BCD system.

DECIMAL	8421 BCD	BINARY
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	0001 0000	1010
11	0001 0001	1011
12	0001 0010	1100
13	0001 0011	1101
14	0001 0100	1110
15	0001 0101	1111

Figure 1-11

Codes.

To represent a decimal number in BCD notation, substitute the appropriate 4-bit code for each decimal digit. For example, the decimal integer 834 in BCD would be 1000 0011 0100. Each decimal digit is represented by its equivalent 8421 4-bit code. A space is left between each 4-bit group to avoid confusing the BCD format with the pure binary code. This method of representation also applies to decimal fractions. For example, the decimal fraction 0.764 would be 0.0111 0110 0100 in BCD. Again, each decimal digit is represented by its equivalent 8421 4-bit code, with a space between each group.

The BCD code simplifies the man-machine interface but it is less efficient than the pure binary code. It takes more bits to represent a given decimal number in BCD than it does with pure binary notation. For example, the decimal number 83 in pure binary form is 1010011. In BCD code the decimal number 83 is written as 1000 0011.

Decimal-to-BCD conversion is simple and straightforward. However, binary-to-BCD conversion is not direct. An intermediate conversion to decimal must be performed first. For example, the binary number 1011.01 is converted into its BCD equivalent.

First the binary number is converted to decimal.

$$\begin{aligned} 1011.01_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned}$$

Then the decimal result is converted to BCD.

$$11.25_{10} = 0001\ 0001.0010\ 0101$$

To convert from BCD to binary, the previous operation is reversed. For example, the BCD number 1001 0110.0110 0010 0101 is converted into its binary equivalent.



First, the BCD number is converted to decimal.

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10}$$

Then the decimal result is converted to binary.

$$96.625_{10} = 96_{10} + 0.625_{10}$$

$96 \div 2 = 48$	with remainder	0	← LSB
$48 \div 2 = 24$		0	
$24 \div 2 = 12$		0	
$12 \div 2 = 6$		0	
$6 \div 2 = 3$		0	
$3 \div 2 = 1$		1	
$1 \div 2 = 0$		1	← MSB

$$96_{10} = 1100000_2$$

$0.625 \times 2 = 1.25 = 0.25$	with overflow	1	← MSB
$0.250 \times 2 = 0.50 = 0.50$		0	
$0.500 \times 2 = 1.00 = 0$		1	← LSB

$$0.625_{10} = 0.101_2$$

$$96.625_{10} = 96_{10} + 0.625_{10} = 1100000_2 + 0.101_2 = 1100000.101_2$$

Therefore:

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10} = 1100000.101_2$$

Because the intermediate decimal number contains both an integer and fraction, each number portion is converted as described under "Binary Number System." The binary sum (integer plus fraction) 1100000.101 is equivalent to the BCD number 1001 0110.0110 0010 0101.

## Alphanumeric Codes

Several binary codes are called alphanumeric codes because they are used to represent characters as well as numbers. The most common of these codes, ASCII, will be discussed here.

**ASCII Code** The American Standard Code for Information Interchange commonly referred to as ASCII, is a special form of binary code that is widely used in microprocessors and data communications equipment. ASCII is binary code that is used in transferring data between microprocessors and their peripheral devices, and in communicating data by radio and telephone. A 7-bit code called full ASCII can be represented by  $2^7=128$  different characters. In addition to the characters and numbers generated by 6-bit ASCII, 7-bit ASCII contains lower-case letters of the alphabet, and additional characters for punctuation and control. The 7-bit ASCII code is shown in Figure 1-12.

COLUMN		0 <sup>(3)</sup>	1 <sup>(3)</sup>	2 <sup>(3)</sup>	3	4	5	6	7 <sup>(3)</sup>
ROW	BITS 4321 765	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	\	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	+	;	K	[	k	{
12	1100	FF	FS	,	<	L	\	l	!
13	1101	CR	GS	-	=	M	]	m	}
14	1110	SO	RS	.	>	N	⌒ <sup>(1)</sup>	n	~
15	1111	SI	US	/	?	O	— <sup>(2)</sup>	o	DEL

Figure 1-12  
Table of 7-bit American Standard Code  
for Information Interchange.



## NOTES:

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) Explanation of special control functions in columns 0, 1, 2, and 7.

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tabulation	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space (blank)	DEL	Delete

Figure 1-12

(Continued.)



The 7-bit ASCII code for each number, letter or control function is made up of a 4-bit group and a 3-bit group. Figure 1-13 shows the arrangement of these two groups and the numbering sequence. The 4-bit group is on the right and bit 1 is the LSB. Note how these groups are arranged in rows and columns in Figure 1-12.

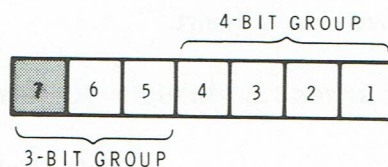


Figure 1-13  
ASCII code word format.

To determine the ASCII code for a given number letter or control operation, locate that item in the table. Then use the 3- and 4-bit codes associated with the row and column in which the item is located. For example, the ASCII code for the letter L is 1001100. It is located in column 4, row 12. The most significant 3-bit group is 100, while the least significant 4-bit group is 1100. When 6-bit ASCII is used, the 3-bit group is reduced to a 2-bit group as shown in Figure 1-14.

In 7-bit ASCII code, an eighth bit is often used as a **parity** or check bit to determine if the data (character) has been transmitted correctly. The value of this bit is determined by the type of parity desired. **Even parity** means the sum of all the 1 bits, including the parity bit, is an even number. For example, if G is the character transmitted, the ASCII code is 1000111. Since four 1's are in the code, the parity bit is 0. The 8-bit code would be written 01000111.

**Odd Parity** means the sum of all the 1 bits, including the parity bit, is an odd number. If the ASCII code for G was transmitted with odd parity, the binary representation would be 11000111.

COLUMN		0	1	2	3
ROW	BITS 4321	10	11	00	01
	65				
0	0000	SP <sup>(3)</sup>	0	@	P
1	0001	!	1	A	Q
2	0010	”	2	B	R
3	0011	#	3	C	S
4	0100	\$	4	D	T
5	0101	%	5	E	U
6	0110	&	6	F	V
7	0111	'	7	G	W
8	1000	(	8	H	X
9	1001	)	9	I	Y
10	1010	*	:	J	Z
11	1011	+	;	K	
12	1100	,	<	L	\
13	1101	-	=	M	]
14	1110	.	>	N	⌒ <sup>(1)</sup>
15	1111	/	?	O	— <sup>(2)</sup>

Figure 1-14  
Table of 6-bit American Standard Code  
for Information Interchange.

NOTES:

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) SP — Space (blank) for machine control.





*Appendix B*

**6808 DATA SHEETS**

Appendix B

PROB DATA SHEET





# MOTOROLA Semiconductors

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

## Advance Information

### MICROPROCESSOR WITH CLOCK

The MC6808 is a monolithic 8-bit microprocessor that contains all the registers and accumulators of the present MC6800 plus an internal clock oscillator and driver on the same chip.

The MC6808 is completely software-compatible with the MC6800 as well as the entire M6800 family of parts. Hence the MC6808 is expandable to 65K words.

This very cost-effective MPU allows the designer to use the MC6808 in consumer as well as industrial applications without sacrificing industrial specifications.

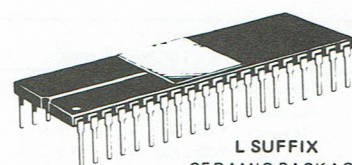
- On-Chip Clock Circuit
- Software-Compatible with the MC6800
- Expandable to 65K words
- Standard TTL-Compatible Inputs and Outputs
- 8-Bit Word Size
- 16-Bit Memory Addressing
- Interrupt Capability

# MC6808

## MOS

(N-CHANNEL, SILICON-GATE,  
DEPLETION LOAD)

### MICROPROCESSOR WITH CLOCK

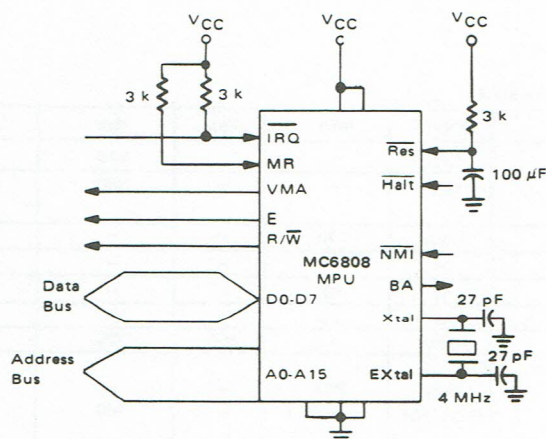


L SUFFIX  
CERAMIC PACKAGE  
CASE 715



P SUFFIX  
PLASTIC PACKAGE  
CASE 711

FIGURE 1 — TYPICAL MICROPROCESSOR INTERFACE



PIN ASSIGNMENT

1	VSS	Reset	40
2	Halt	EXtal	39
3	MR	Xtal	38
4	IRQ	E	37
5	VMA	VSS	36
6	NMI	VCC	35
7	BA	R/W	34
8	VCC	D0	33
9	A0	D1	32
10	A1	D2	31
11	A2	D3	30
12	A3	D4	29
13	A4	D5	28
14	A5	D6	27
15	A6	D7	26
16	A7	A15	25
17	A8	A14	24
18	A9	A13	23
19	A10	A12	22
20	A11	VSS	21



## MC6808

## MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	$V_{CC}$	-0.3 to +7.0	Vdc
Input Voltage	$V_{in}$	-0.3 to +7.0	Vdc
Operating Temperature Range	$T_A$	0 to +70	°C
Storage Temperature Range	$T_{stg}$	-55 to +150	°C
Thermal Resistance	$\theta_{JA}$	100 50	°C/W
	Plastic Ceramic		

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

ELECTRICAL CHARACTERISTICS ( $V_{CC} = 5.0 \text{ V} \pm 5\%$ ,  $V_{SS} = 0$ ,  $T_A = 0$  to  $70^\circ\text{C}$  unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage Logic, EXtal Reset	$V_{IH}$	$V_{SS} + 2.0$ $V_{SS} + 4.0$	—	$V_{CC}$ $V_{CC}$	Vdc
Input Low Voltage Logic, EXtal Reset	$V_{IL}$	$V_{SS} - 0.3$ $V_{SS} - 0.3$	—	$V_{SS} + 0.8$ $V_{SS} + 2.3$	Vdc
Input Leakage Current ( $V_{in} = 0$ to $5.25 \text{ V}$ , $V_{CC} = \text{max}$ )	$I_{in}$	—	1.0	2.5	$\mu\text{Adc}$
Output High Voltage ( $I_{Load} = -205 \mu\text{Adc}$ , $V_{CC} = \text{min}$ ) ( $I_{Load} = -145 \mu\text{Adc}$ , $V_{CC} = \text{min}$ ) ( $I_{Load} = -100 \mu\text{Adc}$ , $V_{CC} = \text{min}$ )	$V_{OH}$	$V_{SS} + 2.4$ $V_{SS} + 2.4$ $V_{SS} + 2.4$	— — —	— — —	Vdc
Output Low Voltage ( $I_{Load} = 1.6 \text{ mAdc}$ , $V_{CC} = \text{min}$ )	$V_{OL}$	—	—	$V_{SS} + 0.4$	Vdc
Power Dissipation	$P_{D^{**}}$	—	0.600	1.2	W
Capacitance # ( $V_{in} = 0$ , $T_A = 25^\circ\text{C}$ , $f = 1.0 \text{ MHz}$ )	$C_{in}$	—	10 6.5	12.5 10	pF
	$C_{out}$	—	—	12	pF
Frequency of Operation (Input Clock $\div 4$ ) (Crystal Frequency)	$f$ $f_{Xtal}$	0.1 1.0	— —	1.0 4.0	MHz
Clock Timing					
Cycle Time	$t_{cyc}$	1.0	—	10	$\mu\text{s}$
Clock Pulse Width (measured at 2.4V)	$PW_{\phi Hs}$	450	—	4500	ns
(measured at 0.4V)	$PW_{\phi L}$	450	—	4500	ns
Fall Time (Measured between $V_{SS} + 0.4 \text{ V}$ and $V_{SS} + 2.4 \text{ V}$ )	$t_{\phi}$	—	—	25	ns

\*Except  $\overline{IRQ}$  and  $\overline{NMI}$ , which require  $3 \text{ k}\Omega$  pullup load resistors for wire-OR capability at optimum operation. Does not include EXtal and Xtal, which are crystal inputs.

#Capacitances are periodically sampled rather than 100% tested.

## READ/WRITE TIMING (Figures 2 through 6; Load Circuit of Figure 4.)

Characteristic	Symbol	Min	Typ	Max	Unit
Address Delay	$t_{AD}$	—	—	270	ns
Peripheral Read Access Time $t_{acc} = t_{ut} - (t_{AD} + t_{DSR})$ ; $t_{ut} = t_{cyc} - t_{\phi}$	$t_{acc}$	—	—	530	ns
Data Setup Time (Read)	$t_{DSR}$	100	—	—	ns
Input Data Hold Time	$t_H$	10	—	—	ns
Output Data Hold Time	$t_H$	30	—	—	ns
Address Hold Time (Address, R/W, VMA)	$t_{AH}$	20	—	—	ns
Data Delay Time (Write)	$t_{PDW}$	—	165	225	ns
Processor Controls					
Processor Control Setup Time	$t_{PCS}$	200	—	—	ns
Processor Control Rise and Fall Time (Measured between 0.8 V and 2.0 V)	$t_{PCr}$ , $t_{PCf}$	—	—	100	ns
Bus Available Delay Time	$t_{BA}$	—	—	250	ns





## MC6808

FIGURE 2 – READ DATA FROM MEMORY OR PERIPHERALS

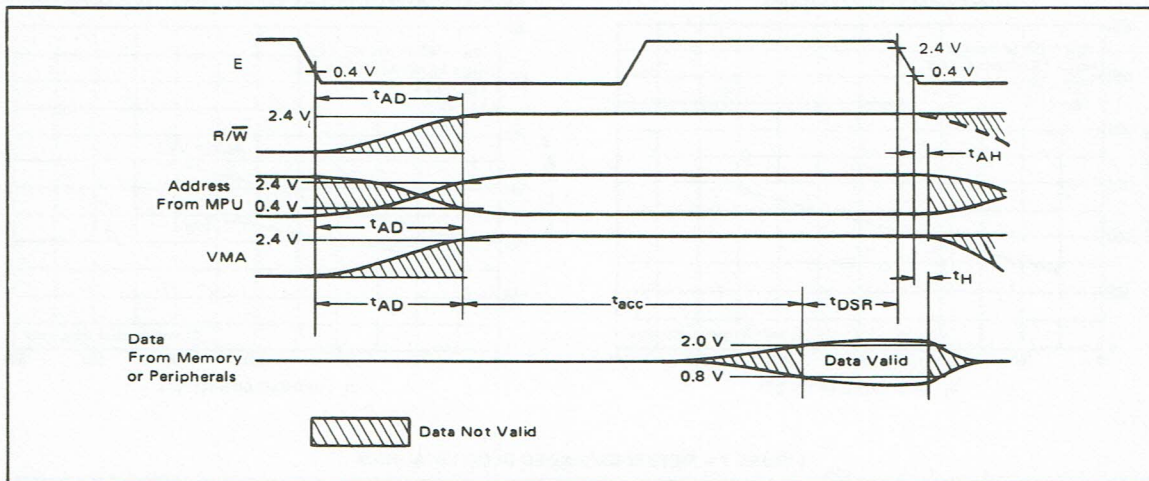


FIGURE 3 – WRITE DATA IN MEMORY OR PERIPHERALS

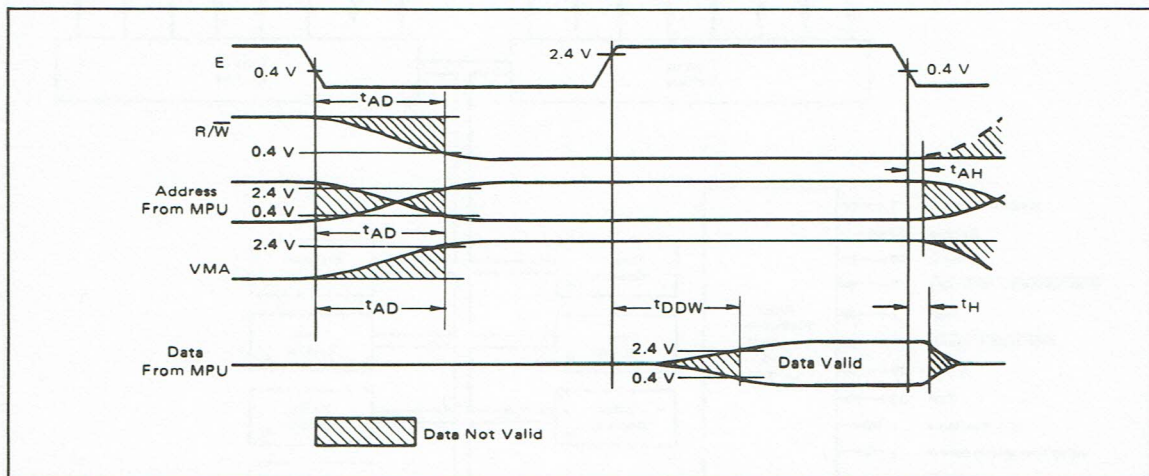
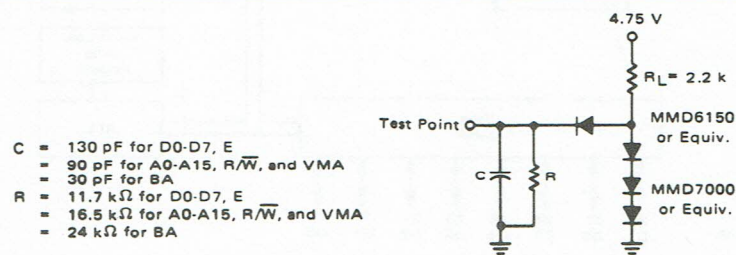
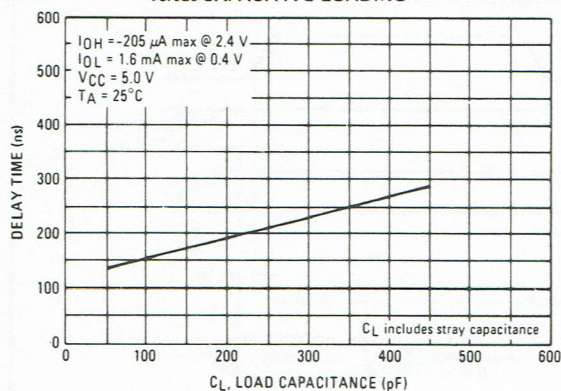


FIGURE 4 – BUS TIMING TEST LOAD

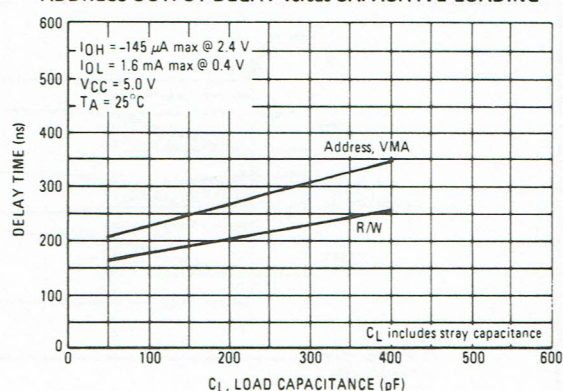


**MC6808**

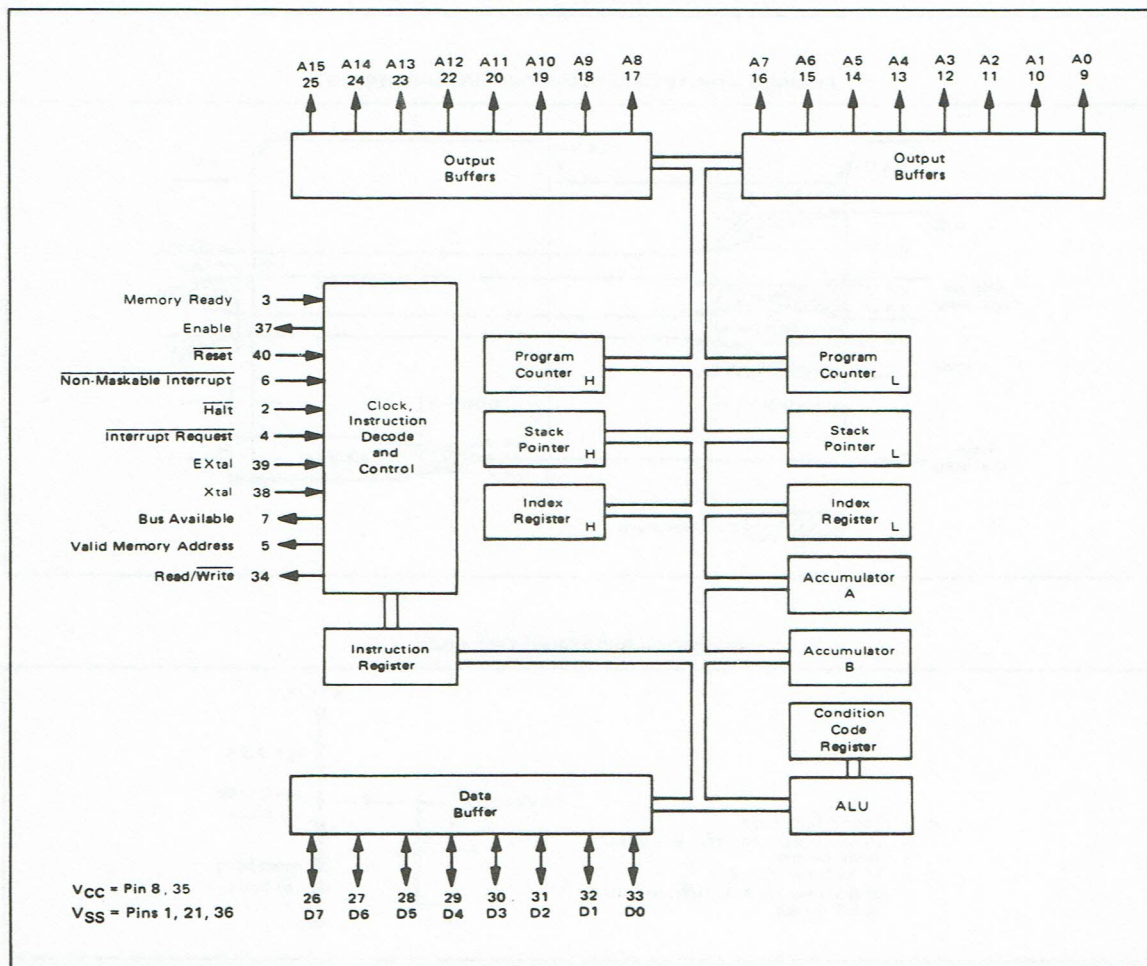
**FIGURE 5 – TYPICAL DATA BUS OUTPUT DELAY  
versus CAPACITIVE LOADING**



**FIGURE 6 – TYPICAL READ/WRITE, VMA, AND  
ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING**



**FIGURE 7 – MC6808 EXPANDED BLOCK DIAGRAM**





## MC6808

## MPU REGISTERS

A general block diagram of the MC6808 is shown in Figure 7. As shown, the number and configuration of the registers are the same as for the MC6800.

The MPU has three 16-bit registers and three 8-bit registers available for use by the programmer (Figure 8).

**Program Counter** — The program counter is a two byte (16-bits) register that points to the current program address.

**Stack Pointer** — The stack pointer is a two byte register that contains the address of the next available location in an external push-down/pop-up stack. This stack is normally a random access Read/Write memory that may have any location (address) that is convenient. In those applications that require storage of information in the stack when power is lost, the stack must be non-volatile.

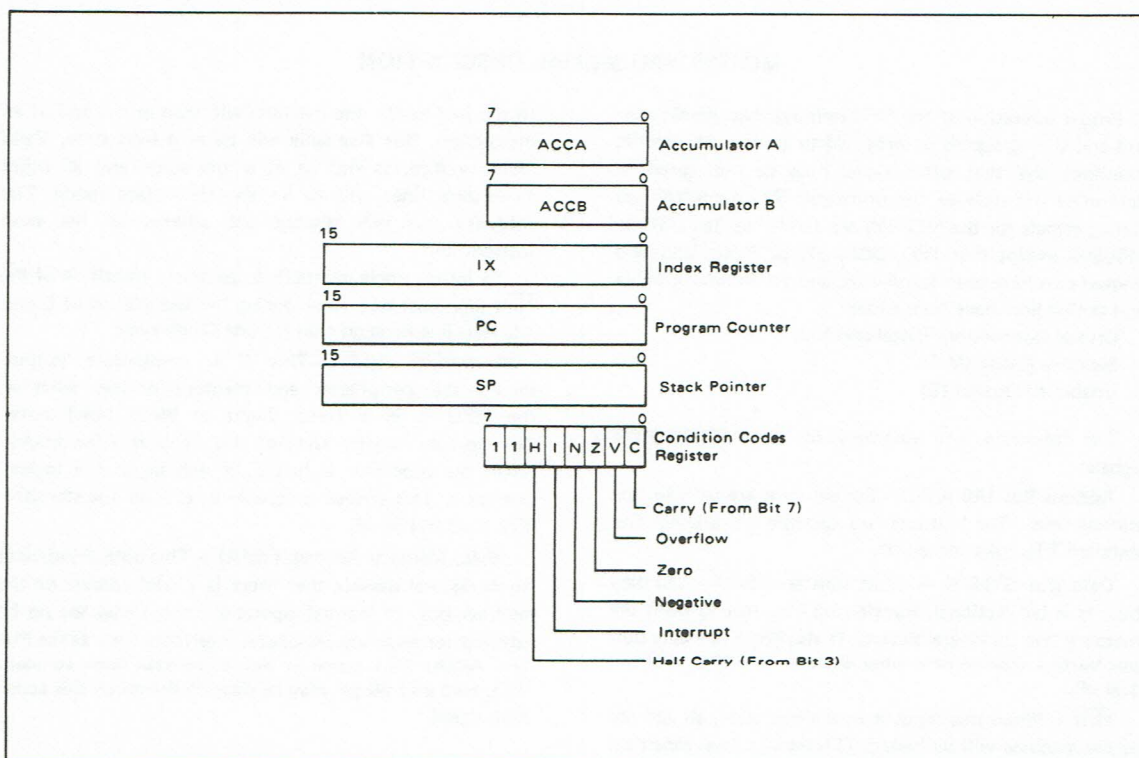
**Index Register** — The index register is a two byte register that is used to store data or a sixteen bit memory address for the Indexed mode of memory addressing.

**Accumulators** — The MPU contains two 8-bit accumulators that are used to hold operands and results from an arithmetic logic unit (ALU).

**Condition Code Register** — The condition code register indicates the results of an Arithmetic Logic Unit operation: Negative (N), Zero (Z), Overflow (V), Carry from bit 7 (C), and half carry from bit 3 (H). These bits of the Condition Code Register are used as testable conditions for the conditional branch instructions. Bit 4 is the interrupt mask bit (I). The used bits of the Condition Code Register (b6 and b7) are ones.

Figure 9 shows the order of saving the microprocessor status within the stack.

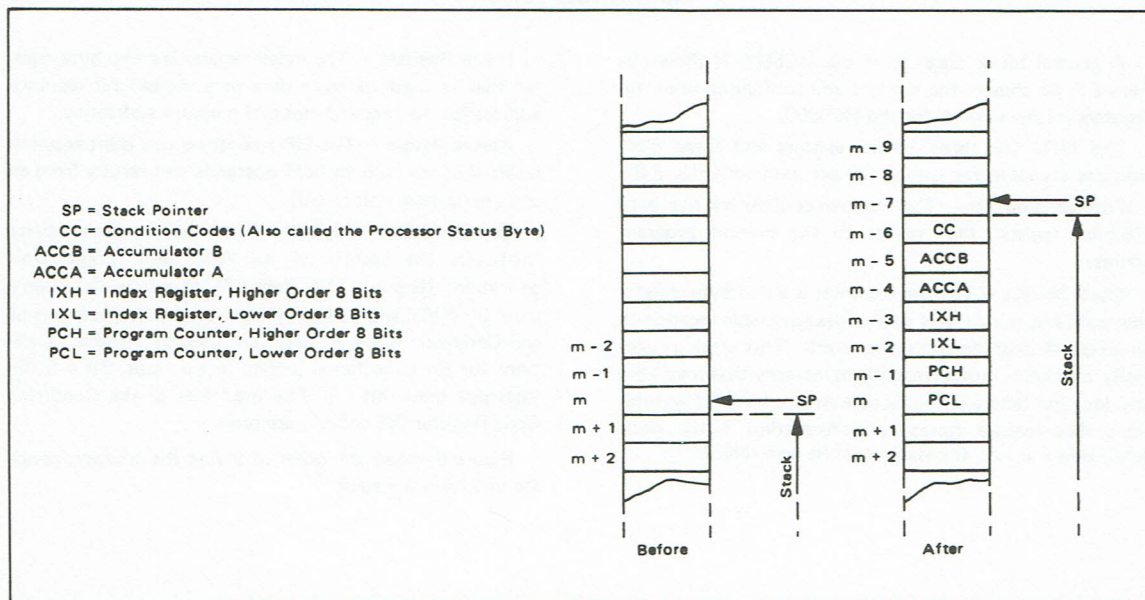
FIGURE 8 — PROGRAMMING MODEL OF THE MICROPROCESSING UNIT





## MC6808

FIGURE 9 — SAVING THE STATUS OF THE MICROPROCESSOR IN THE STACK



## MC6808 MPU SIGNAL DESCRIPTION

Proper operation of the MPU requires that certain control and timing signals be provided to accomplish specific functions and that other signal lines be monitored to determine the state of the processor. These control and timing signals for the MC6808 are similar to those of the MC6800 except that TSC, DBE,  $\phi 1$ ,  $\phi 2$  input, and two unused pins have been eliminated, and the following signal and timing lines have been added:

Crystal Connections Extal and Xtal  
 Memory Ready (MR)  
 Enable  $\phi 2$  Output (E)

The following is a summary of the MC6808 MPU signals:

**Address Bus (A0-A15)** — Sixteen pins are used for the address bus. The outputs are capable of driving one standard TTL load and 90 pF.

**Data Bus (D0-D7)** — Eight pins are used for the data bus. It is bidirectional, transferring data to and from the memory and peripheral devices. It also has three-state output buffers capable of driving one standard TTL load and 130 pF.

**Halt** — When this input is in the low state, all activity in the machine will be halted. This input is level sensitive.

In the halt mode, the machine will stop at the end of an instruction, Bus Available will be at a high state, Valid Memory Address will be at a low state, and all other three-state lines will be in the three-state mode. The address bus will display the address of the next instruction.

To insure single instruction operation, transition of the Halt line must not occur during the last 200 ns of E and the Halt line must go high for one Clock cycle.

**Read/Write (R/W)** — This TTL compatible output signals the peripherals and memory devices whether the MPU is in a Read (high) or Write (low) state. The normal standby state of this signal is Read (high). When the processor is halted, it will be in the logical one state. This output is capable of driving one standard TTL load and 90 pF.

**Valid Memory Address (VMA)** — This output indicates to peripheral devices that there is a valid address on the address bus. In normal operation, this signal should be utilized for enabling peripheral interfaces such as the PIA and ACIA. This signal is not three-state. One standard TTL load and 90 pF may be directly driven by this active high signal.





## MC6808

**Bus Available (BA)** — The Bus Available signal will normally be in the low state; when activated, it will go to the high state indicating that the microprocessor has stopped and that the address bus is available but not in three-state. This will occur if the  $\overline{\text{Halt}}$  line is in the low state or the processor is in the WAIT state as a result of the execution of a WAIT instruction. At such time, all three-state output drivers will go to their off state and other outputs to their normally inactive level. The processor is removed from the WAIT state by the occurrence of a maskable (mask bit  $I = 0$ ) or nonmaskable interrupt. This output is capable of driving one standard TTL load and 30 pF.

**Interrupt Request ( $\overline{\text{IRQ}}$ )** — This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further interrupts may occur. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations

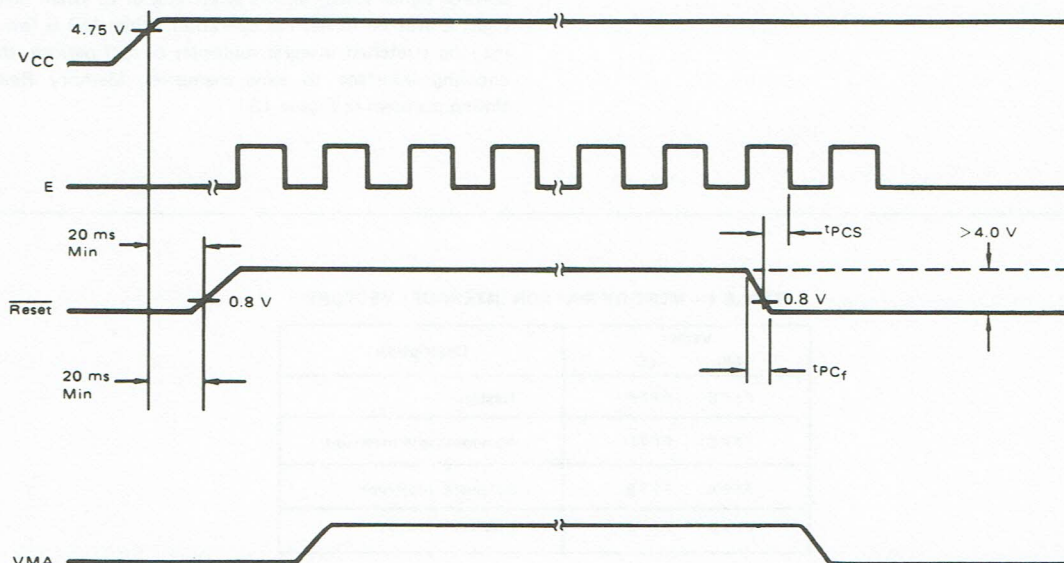
FFF8 and FFF9. An address loaded at these locations causes the MPU to branch to an interrupt routine in memory.

The  $\overline{\text{Halt}}$  line must be in the high state for interrupts to be serviced. Interrupts will be latched internally while  $\overline{\text{Halt}}$  is low.

The  $\overline{\text{IRQ}}$  has a high impedance pullup device internal to the chip; however a 3 k $\Omega$  external resistor to  $V_{CC}$  should be used for wire-OR and optimum control of interrupts.

**Reset** — This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. When this line is low, the MPU is inactive and the information in the registers will be lost. If a high level is detected on the input, this will signal the MPU to begin the restart sequence. This will start execution of a routine to initialize the processor from its reset condition. All the higher order address lines will be forced high. For the restart, the last two (FFFE, FFFF) locations in memory will be used to load the program that is addressed by the program counter. During the restart routine, the interrupt mask bit is set and must be reset before the MPU can be interrupted by  $\overline{\text{IRQ}}$ . Power-up and reset timing sequences are shown in Figure 10.

FIGURE 10 — POWER-UP AND RESET TIMING

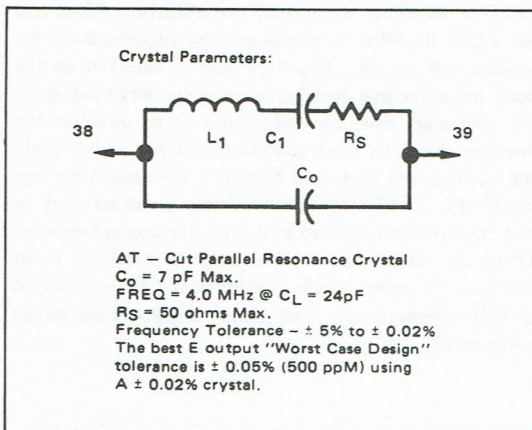




## MC6808

**EXtal and Xtal** — The MC 6808 has an internal oscillator that may be crystal controlled. These connections are for a parallel resonant fundamental crystal. (AT cut.) A divide-by-four circuit has been added to the MC6808 so that a 4 MHz crystal may be used in lieu of a 1 MHz crystal for a more cost-effective system. Pin 39 of the MC6808 may be driven externally by a TTL input signal if a separate clock is required. Pin 38 is to be left open in this mode. Crystal parameters to be specified are in Figure 11.

FIGURE 11—CRYSTAL PARAMETERS



**Non-Maskable Interrupt (NMI)** — A low-going edge on this input requests that a non-mask-interrupt sequence be generated within the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the NMI signal. The interrupt mask bit in the Condition Code Register has no effect on NMI.

The index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations FFFC and FFFD. An address loaded at these locations caused the MPU to branch to a non-maskable interrupt routine in memory.

NMI has a high impedance pullup resistor internal to the chip; however a  $3 \text{ k}\Omega$  external resistor to  $V_{CC}$  should be used for wire-OR and optimum control of interrupts.

Inputs IRQ and NMI are hardware interrupt lines that are sampled when E is high and will start the interrupt routine on a low E following the completion of an instruction.

Figure 12 is a flow chart describing the major decision paths and interrupt vectors of the microprocessor. Table 1 gives the memory map for interrupt vectors.

**Memory Ready (MR)** — MR is a TTL compatible input control signal which allows stretching of E. When MR is high, E will be in normal operation. When MR is low, E may be stretched integral multiples of half periods, thus allowing interface to slow memories. Memory Ready timing is shown in Figure 13.

TABLE 1 — MEMORY MAP FOR INTERRUPT VECTORS

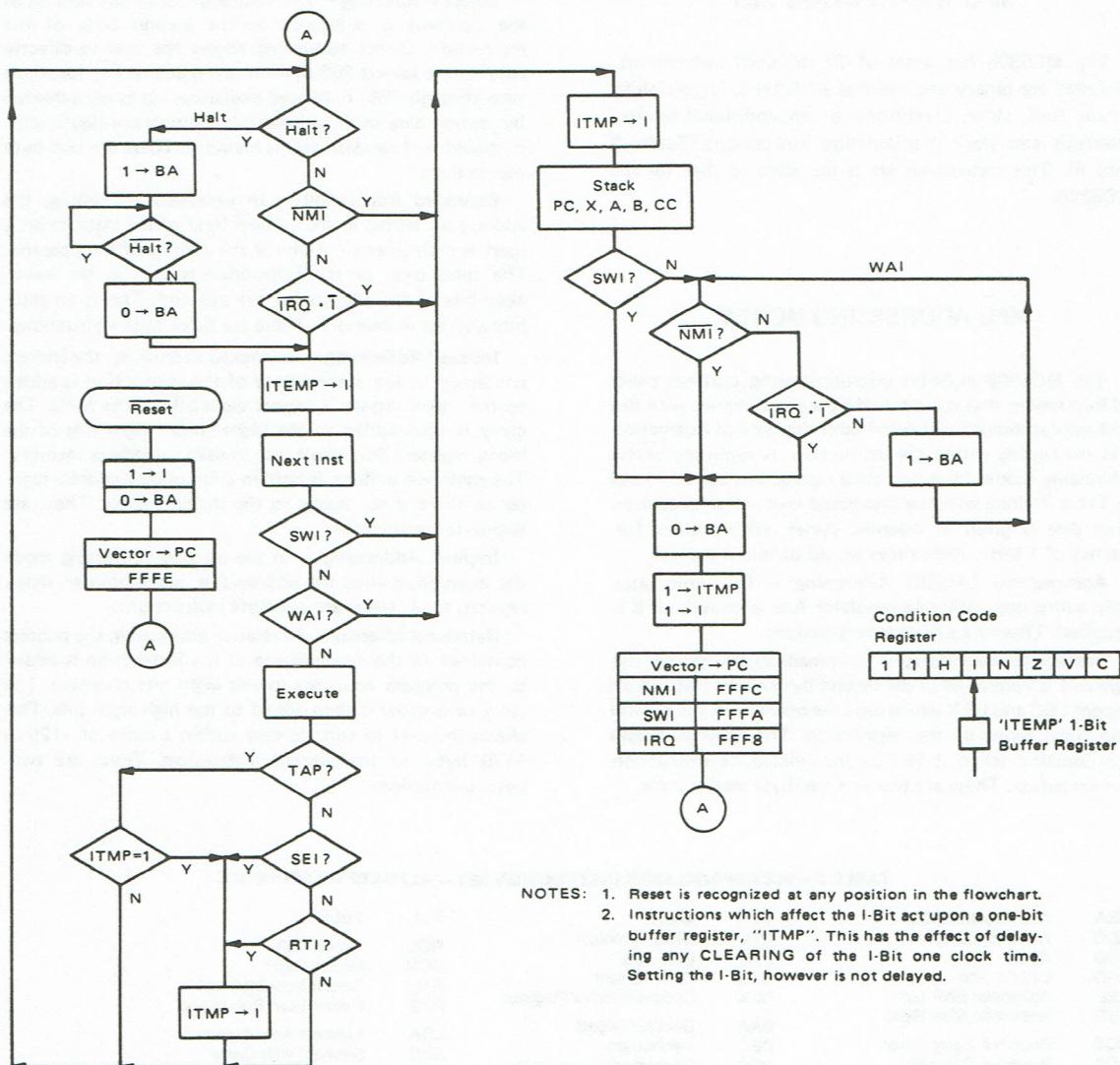
Vector		Description
MS	LS	
FFFE	FFFF	Restart
FFFC	FFFD	Non-maskable Interrupt
FFFA	FFFB	Software Interrupt
FFF8	FFF9	Interrupt Request



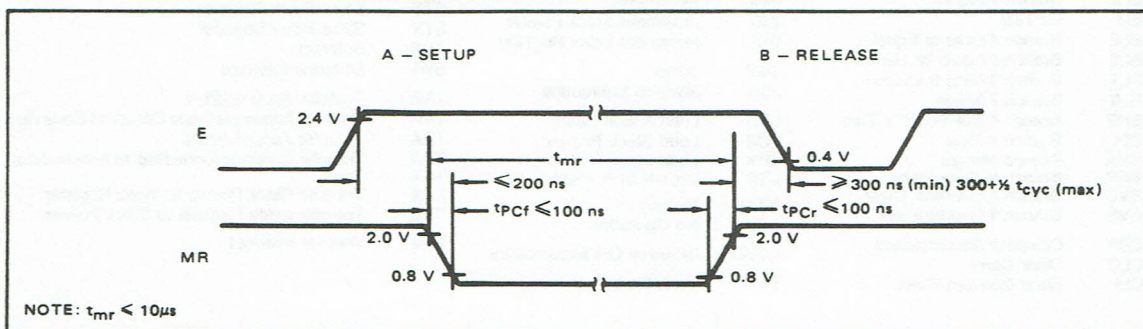


MC6808

FIGURE 12 – MPU FLOW CHART



### FIGURE 13 – MEMORY READY CONTROL FUNCTION





## MC6808

## MPU INSTRUCTION SET

The MC6808 has a set of 72 different instructions. Included are binary and decimal arithmetic, logical, shift, rotate, load, store, conditional or unconditional branch, interrupt and stack manipulation instructions (Tables 2 thru 6). This instruction set is the same as that for the MC6800.

## MPU ADDRESSING MODES

The MC6808 eight-bit microprocessing unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 7 along with the associated instruction execution time that is given in machine cycles. With a clock frequency of 1 MHz, these times would be microseconds.

**Accumulator (ACCX) Addressing** — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one-byte instructions.

**Immediate Addressing** — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MPU addresses this location when it fetches the immediate instruction for execution. These are two or three-byte instructions.

**Direct Addressing** — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

**Extended Addressing** — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight-bits of the address of the operand. The third byte of the instruction is used as the lower eight-bits of the address for the operand. This is an absolute address in memory. These are three-byte instructions.

**Indexed Addressing** — In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits in the MPU. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

**Implied Addressing** — In the implied addressing mode the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

**Relative Addressing** — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two-byte instructions.

TABLE 2 — MICROPROCESSOR INSTRUCTION SET — ALPHABETIC SEQUENCE

ABA	Add Accumulators	CLR	Clear	PUL	Pull Data
ADC	Add with Carry	CLV	Clear Overflow	ROL	Rotate Left
ADD	Add	CMP	Compare	ROR	Rotate Right
AND	Logical And	COM	Complement	RTI	Return from Interrupt
ASL	Arithmetic Shift Left	CPX	Compare Index Register	RTS	Return from Subroutine
ASR	Arithmetic Shift Right	DAA	Decimal Adjust	SBA	Subtract Accumulators
BCC	Branch if Carry Clear	DEC	Decrement	SBC	Subtract with Carry
BCS	Branch if Carry Set	DES	Decrement Stack Pointer	SEC	Set Carry
BEQ	Branch if Equal to Zero	DEX	Decrement Index Register	SEI	Set Interrupt Mask
BGE	Branch if Greater or Equal Zero	EOR	Exclusive OR	SEV	Set Overflow
BGT	Branch if Greater than Zero	INC	Increment	STA	Store Accumulator
BHI	Branch if Higher	INS	Increment Stack Pointer	STS	Store Stack Register
BIT	Bit Test	INX	Increment Index Register	STX	Store Index Register
BLE	Branch if Less or Equal	JMP	Jump	SUB	Subtract
BLS	Branch if Lower or Same	JSR	Jump to Subroutine	SWI	Software Interrupt
BLT	Branch if Less than Zero	LDA	Load Accumulator	TAB	Transfer Accumulators
BMI	Branch if Minus	LDS	Load Stack Pointer	TAP	Transfer Accumulators to Condition Code Reg.
BNE	Branch if Not Equal to Zero	LDX	Load Index Register	TBA	Transfer Accumulators
BPL	Branch if Plus	LSR	Logical Shift Right	TPA	Transfer Condition Code Reg. to Accumulator
BRA	Branch Always	NEG	Negate	TST	Test
BSR	Branch to Subroutine	NOP	No Operation	TSX	Transfer Stack Pointer to Index Register
BVC	Branch if Overflow Clear	ORA	Inclusive OR Accumulator	TXS	Transfer Index Register to Stack Pointer
BVS	Branch if Overflow Set	PSH	Push Data	WAI	Wait for Interrupt
CBA	Compare Accumulators				
CLC	Clear Carry				
CLI	Clear Interrupt Mask				





## MC6808

TABLE 3 — ACCUMULATOR AND MEMORY INSTRUCTIONS

ADDRESSING MODES										BOOLEAN/ARITHMETIC OPERATION										COND. CODE REG.				
OPERATIONS	MNEMONIC	IMMED		DIRECT		INDEX		EXTND		IMPLIED		(All register labels refer to contents)						5	4	3	2	1	0	
		OP	=	OP	=	OP	=	OP	=	OP	=		H	I	N	Z	V	C						
Add	ADDA	3B	2 2	9B	3 2	A8	5 2	8B	4 3			A + M - A												
	ADDB	CB	2 2	DB	3 2	E8	5 2	FB	4 3			B + M - B												
Add Acmltrs	ABA									1B	2 1	A + B -> A												
Add with Carry	ADCA	89	2 2	99	3 2	A9	5 2	89	4 3			A + M + C - A												
	ADCB	C9	2 2	D9	3 2	E9	5 2	F9	4 3			B + M + C - B												
And	ANDA	84	2 2	94	3 2	A4	5 2	84	4 3			A - M - A												
	ANDB	C4	2 2	D4	3 2	E4	5 2	F4	4 3			B - M - B												
Bit Test	BITA	85	2 2	95	3 2	A5	5 2	85	4 3			A - M												
	BITB	C5	2 2	D5	3 2	E5	5 2	F5	4 3			B - M												
Clear	CLR					6F	7 2	7F	6 3			00 - M												
	CLRA									4F	2 1	00 -> A												
	CLRB									5F	2 1	00 -> B												
Compare	CMPA	81	2 2	91	3 2	A1	5 2	81	4 3			A - M												
	CMPB	C1	2 2	D1	3 2	E1	5 2	F1	4 3			B - M												
Compare Acmltrs	CBA									11	2 1	A - B												
Complement, 1's	COM					63	7 2	73	6 3			M - M												
	COMA									43	2 1	A -> A												
	COMB									53	2 1	B -> B												
Complement, 2's (Negate)	NEG					60	7 2	70	6 3			00 - M - M												
	NEGA									40	2 1	00 -> A -> A												
	NEGB									50	2 1	00 -> B -> B												
Decimal Adjust, A	DAA									19	2 1	Converts Binary Add. of BCD Characters into BCD Format												
Decrement	DEC					6A	7 2	7A	6 3			M - 1 - M												
	DECA									4A	2 1	A - 1 - A												
	DECB									5A	2 1	B - 1 - B												
Exclusive OR	EORA	88	2 2	98	3 2	A8	5 2	88	4 3			A ⊕ M - A												
	EORB	C8	2 2	D8	3 2	E8	5 2	F8	4 3			B ⊕ M - B												
Increment	INC					6C	7 2	7C	6 3			M + 1 - M												
	INCA									4C	2 1	A + 1 - A												
	INCB									5C	2 1	B + 1 - B												
Load Acmltr	LDAA	86	2 2	96	3 2	A6	5 2	86	4 3			M -> A												
	LDAB	C6	2 2	D6	3 2	E6	5 2	F6	4 3			M -> B												
Or, Inclusive	ORA	8A	2 2	9A	3 2	AA	5 2	8A	4 3			A + M -> A												
	ORB	CA	2 2	DA	3 2	EA	5 2	FA	4 3			B + M -> B												
Push Data	PSHA									36	4 1	A -> Msp, SP - 1 - SP												
	PSHB									37	4 1	B -> Msp, SP - 1 - SP												
Pull Data	PULA									32	4 1	SP + 1 - SP, Msp -> A												
	PULB									33	4 1	SP + 1 - SP, Msp -> B												
Rotate Left	ROL					69	7 2	79	6 3			M												
	ROLA									49	2 1	A												
	ROLB									59	2 1	B												
Rotate Right	ROR					66	7 2	76	6 3			M												
	RORA									46	2 1	A												
	RORB									56	2 1	B												
Shift Left, Arithmetic	ASL					68	7 2	78	6 3			M												
	ASLA									48	2 1	A												
	ASLB									58	2 1	B												
Shift Right, Arithmetic	ASR					67	7 2	77	6 3			M												
	ASRA									47	2 1	A												
	ASRB									57	2 1	B												
Shift Right, Logic	LSR					64	7 2	74	6 3			M												
	LSRA									44	2 1	A												
	LSRB									54	2 1	B												
Store Acmltr	STAA			97	4 2	A7	6 2	B7	5 3			A -> M												
	STAB			07	4 2	E7	6 2	F7	5 3			B -> M												
Subtract	SUBA	80	2 2	90	3 2	A0	5 2	80	4 3			A - M -> A												
	SUBB	C0	2 2	D0	3 2	E0	5 2	F0	4 3			B - M -> B												
Subtract Acmltrs	SBA									10	2 1	A - B -> A												
Subtr. with Carry	SBCA	82	2 2	92	3 2	A2	5 2	82	4 3			A - M - C -> A												
	SBCB	C2	2 2	D2	3 2	E2	5 2	F2	4 3			B - M - C -> B												
Transfer Acmltrs	TAB									16	2 1	A -> B												
	TBA									17	2 1	B -> A												
Test, Zero or Minus	TST					60	7 2	70	6 3			M -> 00												
	TSTA									40	2 1	A -> 00												
	TSTB									50	2 1	B -> 00												

## LEGEND:

OP Operation Code (Hexadecimal);  
 ~ Number of MPU Cycles;  
 = Number of Program Bytes;  
 + Arithmetic Plus;  
 - Arithmetic Minus;  
 \* Boolean AND;

Msp Contents of memory location pointed to by Stack Pointer;

+ Boolean Inclusive OR;  
 ⊖ Boolean Exclusive OR;  
 ⊖ Complement of M;  
 → Transfer Into;  
 0 Bit = Zero;  
 00 Byte = Zero;

## CONDITION CODE SYMBOLS:

H Half carry from bit 3;  
 I Interrupt mask;  
 N Negative (sign bit);  
 Z Zero (byte);  
 V Overflow, 2's complement;  
 C Carry from bit 7;  
 R Reset Always;  
 S Set Always;  
 : Test and set if true, cleared otherwise;  
 ● Not Affected

Note — Accumulator addressing mode instructions are included in the column for IMPLIED addressing



MOTOROLA Semiconductor Products Inc.



MC6808

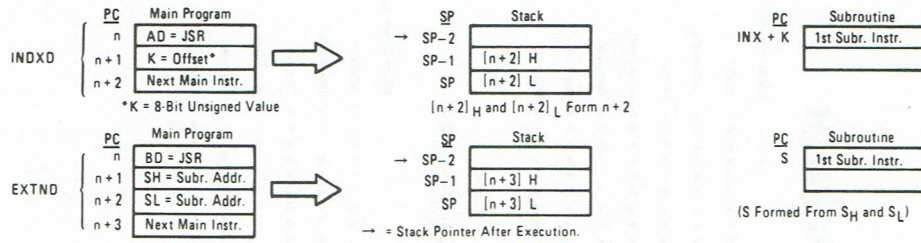
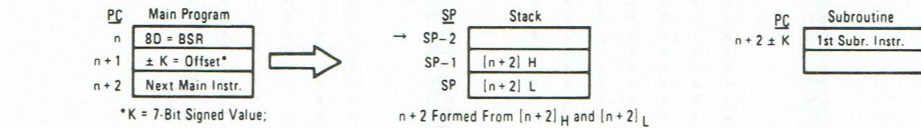
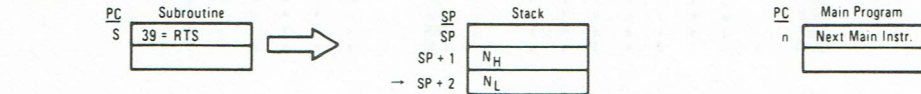
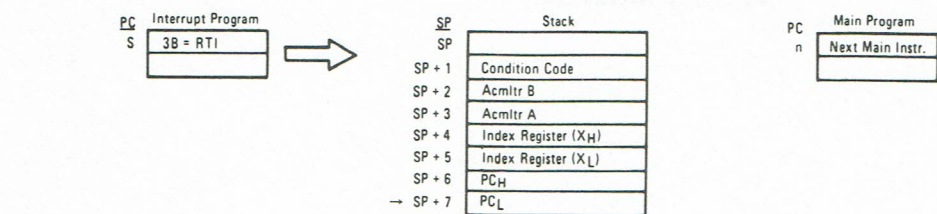
#### TABLE 4 – INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

																	COND. CODE REG.																
		IMMED			DIRECT			INDEX			EXTND			IMPLIED																			
POINTER OPERATIONS	MNEMONIC	OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	=	BOOLEAN/ARITHMETIC OPERATION	H	I	N	Z	V	C										
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$X_H - M, X_L \rightarrow (M + 1)$	•	•	⑦	:	⑧											
Decrement Index Reg	DEX													09	4	1	$X \rightarrow X - 1$	•	•	•	•	•											
Decrement Stack Ptr	DES													34	4	1	$SP \rightarrow SP - 1$	•	•	•	•	•											
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	:	•											
Increment Stack Ptr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•											
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	⑨	:	R											
Load Stack Ptr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	⑨	:	R											
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	⑨	:	R											
Store Stack Ptr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	⑨	:	R											
Index Reg $\rightarrow$ Stack Ptr	TXS													35	4	1	$X \rightarrow SP - 1$	•	•	•	•	•											
Stack Ptr $\rightarrow$ Index Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•											

### TABLE 5 – JUMP AND BRANCH INSTRUCTIONS

															COND. CODE REG.							
		RELATIVE			INDEX			EXTND			IMPLIED											
OPERATIONS		MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	BRANCH TEST							
															5	4	3	2	1	0		
															H	I	N	Z	V	C		
Branch Always	BRA	20	4	2											None	•	•	•	•	•	•	
Branch If Carry Clear	BCC	24	4	2											C = 0	•	•	•	•	•	•	
Branch If Carry Set	BCS	25	4	2											C = 1	•	•	•	•	•	•	
Branch If = Zero	BEQ	27	4	2											Z = 1	•	•	•	•	•	•	
Branch If ≥ Zero	BGE	2C	4	2											$N \oplus V = 0$	•	•	•	•	•	•	
Branch If > Zero	BGT	2E	4	2											$Z + (N \oplus V) = 0$	•	•	•	•	•	•	
Branch If Higher	BHI	22	4	2											C + Z = 0	•	•	•	•	•	•	
Branch If ≤ Zero	BLE	2F	4	2											$Z + (N \oplus V) = 1$	•	•	•	•	•	•	
Branch If Lower Or Same	BLS	23	4	2											C + Z = 1	•	•	•	•	•	•	
Branch If < Zero	BLT	2D	4	2											$N \oplus V = 1$	•	•	•	•	•	•	
Branch If Minus	BMI	2B	4	2											N = 1	•	•	•	•	•	•	
Branch If Not Equal Zero	BNE	26	4	2											Z = 0	•	•	•	•	•	•	
Branch If Overflow Clear	BVC	28	4	2											V = 0	•	•	•	•	•	•	
Branch If Overflow Set	BVS	29	4	2											V = 1	•	•	•	•	•	•	
Branch If Plus	BPL	2A	4	2											N = 0	•	•	•	•	•	•	
Branch To Subroutine	BSR	8D	8	2												•	•	•	•	•	•	
Jump	JMP					6E	4	2	7E	3	3				} See Special Operations	•	•	•	•	•	•	
Jump To Subroutine	JSR					AD	8	2	8D	9	3					•	•	•	•	•	•	•
No Operation	NOP											01	2	1	} Advances Prog. Cntr. Only	•	•	•	•	•	•	
Return From Interrupt	RTI											3B	10	1		•	•	•	•	•	•	•
Return From Subroutine	RTS											39	5	1		•	•	•	•	•	•	•
Software Interrupt	SWI											3F	12	1	} See Special Operations	•	•	•	•	•	•	
Wait for Interrupt*	WAI											3E	9	1		•	•	•	•	•	•	•

\*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

**MC6808****SPECIAL OPERATIONS****JSR, JUMP TO SUBROUTINE:****BSR, BRANCH TO SUBROUTINE:****JMP, JUMP:****RTS, RETURN FROM SUBROUTINE:****RTI, RETURN FROM INTERRUPT:****TABLE 6 – CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS**

OPERATIONS	MNEMONIC	IMPLIED			BOOLEAN OPERATION	COND. CODE REG.						
		OP	~	=		5	4	3	2	1	0	
						H	I	N	Z	V	C	
Clear Carry	CLC	0C	2	1	0 → C	•	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	0 → I	•	R	•	•	•	•	•
Clear Overflow	CLV	0A	2	1	0 → V	•	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	1 → C	•	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	1 → I	•	S	•	•	•	•	•
Set Overflow	SEV	0B	2	1	1 → V	•	•	•	•	•	•	S
Accmltr A → CCR	TAP	06	2	1	A → CCR	•	•	•	•	•	•	•
CCR → Accmltr A	TPA	07	2	1	CCR → A	•	•	•	•	•	•	•

CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

- |   |   |
|---|---|
| 1 (Bit V) Test: Result = 10000000?  | 7 (Bit N) Test: Sign bit of most significant (MS) byte = 1?   |
| 2 (Bit C) Test: Result ≠ 00000000?  | 8 (Bit V) Test: 2's complement overflow from subtraction of MS bytes?   |
| 3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.) | 9 (Bit N) Test: Result less than zero? (Bit 15 = 1)   |
| 4 (Bit V) Test: Operand = 10000000 prior to execution?  | 10 (All) Load Condition Code Register from Stack. (See Special Operations)  |
| 5 (Bit V) Test: Operand = 01111111 prior to execution?  | 11 (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state. |
| 6 (Bit V) Test: Set equal to result of N⊕C after shift has occurred.  | 12 (All) Set according to the contents of Accumulator A.  |





MC6808

TABLE 7 – INSTRUCTION ADDRESSING MODES AND ASSOCIATED EXECUTION TIMES  
(Times in Machine Cycles)

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied	Relative		(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied
ABA		•	•	•	•	•	2	•	INC	2	•	•	•	•	•	•
ADC	x	•	2	3	4	5	•	•	INS	•	•	•	•	•	•	•
ADD	x	•	2	3	4	5	•	•	INX	•	•	•	•	•	•	•
AND	x	•	2	3	4	5	•	•	JMP	•	•	•	•	•	•	•
ASL	2	•	•	•	6	7	•	•	JSR	•	•	•	•	•	•	•
ASR	2	•	•	•	6	7	•	•	LDA	x	2	3	4	5	•	•
BCC	•	•	•	•	•	•	•	4	LDS	•	•	•	•	•	•	•
BCS	•	•	•	•	•	•	•	4	LDX	•	•	•	•	•	•	•
BEA	•	•	•	•	•	•	•	4	LSR	•	•	•	•	•	•	•
BGE	•	•	•	•	•	•	•	4	NEG	•	•	•	•	•	•	•
BGT	•	•	•	•	•	•	•	4	NOP	•	•	•	•	•	•	•
BHI	•	•	•	•	•	•	•	4	ORA	x	2	3	4	5	•	•
BIT	x	•	2	3	4	5	•	•	PSH	•	•	•	•	•	•	•
BLE	•	•	•	•	•	•	•	4	PUL	•	•	•	•	•	•	•
BLS	•	•	•	•	•	•	•	4	ROL	•	•	•	•	•	•	•
BLT	•	•	•	•	•	•	•	4	ROR	•	•	•	•	•	•	•
BMI	•	•	•	•	•	•	•	4	RTI	•	•	•	•	•	•	10
BNE	•	•	•	•	•	•	•	4	RTS	•	•	•	•	•	•	5
BPL	•	•	•	•	•	•	•	4	SBA	•	•	•	•	•	•	2
BRA	•	•	•	•	•	•	•	4	SBC	x	2	3	4	5	•	•
BSR	•	•	•	•	•	•	•	8	SEC	•	•	•	•	•	•	•
BVC	•	•	•	•	•	•	•	4	SEI	•	•	•	•	•	•	•
BVS	•	•	•	•	•	•	•	4	SEV	•	•	•	•	•	•	•
CBA	•	•	•	•	•	•	2	•	STA	x	•	4	5	6	•	•
CLC	•	•	•	•	•	•	2	•	STS	•	•	•	•	•	•	•
CLI	•	•	•	•	•	•	2	•	STX	•	•	•	•	•	•	•
CLR	2	•	•	•	6	7	•	•	SUB	x	•	2	3	4	5	•
CLV	•	•	•	•	•	•	2	•	SWI	•	•	•	•	•	•	12
CMP	x	•	2	3	4	5	•	•	TAB	•	•	•	•	•	•	•
COM	•	2	•	•	6	7	•	•	TAP	•	•	•	•	•	•	•
CPX	•	•	3	4	5	6	•	•	TBA	•	•	•	•	•	•	•
DAA	•	•	•	•	•	•	2	•	TPA	•	•	•	•	•	•	•
DEC	2	•	•	•	6	7	•	•	TST	2	•	•	•	6	7	•
DES	•	•	•	•	•	•	4	•	TSX	•	•	•	•	•	•	•
DEX	•	•	•	•	•	•	4	•	TSX	•	•	•	•	•	•	•
EOR	x	•	2	3	4	5	•	•	WAI	•	•	•	•	•	•	9

NOTE Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.





MC6808

## SUMMARY OF CYCLE BY CYCLE OPERATION

Table 8 provides a detailed description of the information present on the Address Bus, Data Bus, Valid Memory Address line (VMA), and the Read/Write line (R/W) during each cycle for each instruction.

This information is useful in comparing actual with expected results during debug of both software and hardware as the control program is executed. The information is categorized in groups according to Addressing Mode and Number of Cycles per instruction. (In general, instructions with the same Addressing Mode and Number of Cycles execute in the same manner; exceptions are indicated in the table.)

ware as the control program is executed. The information is categorized in groups according to Addressing Mode and Number of Cycles per instruction. (In general, instructions with the same Addressing Mode and Number of Cycles execute in the same manner; exceptions are indicated in the table.)

TABLE 8 — OPERATION SUMMARY

TABLE 3 - OPERATION SUMMARY							
Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus	
IMMEDIATE							
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	2	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Operand Data	
CPX LDS LDX	3	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Operand Data (High Order Byte)	
		3	1	Op Code Address + 2	1	Operand Data (Low Order Byte)	
DIRECT							
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	3	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Operand	
		3	1	Address of Operand	1	Operand Data	
CPX LDS LDX	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Operand	
		3	1	Address of Operand	1	Operand Data (High Order Byte)	
		4	1	Operand Address + 1	1	Operand Data (Low Order Byte)	
STA	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Destination Address	
		3	0	Destination Address	1	Irrelevant Data (Note 1)	
		4	1	Destination Address	0	Data from Accumulator	
STS STX	5	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Operand	
		3	0	Address of Operand	1	Irrelevant Data (Note 1)	
		4	1	Address of Operand	0	Register Data (High Order Byte)	
		5	1	Address of Operand + 1	0	Register Data (Low Order Byte)	
INDEXED							
JMP	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Offset	
		3	0	Index Register	1	Irrelevant Data (Note 1)	
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)	
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	5	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Offset	
		3	0	Index Register	1	Irrelevant Data (Note 1)	
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)	
		5	1	Index Register Plus Offset	1	Operand Data	
CPX LDS LDX	6	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Offset	
		3	0	Index Register	1	Irrelevant Data (Note 1)	
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)	
		5	1	Index Register Plus Offset	1	Operand Data (High Order Byte)	
		6	1	Index Register Plus Offset + 1	1	Operand Data (Low Order Byte)	





MC6808

TABLE 8 – OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INDEXED (Continued)						
STA	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	1	Index Register Plus Offset	1	Current Operand Data
		6	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		7	1/0 (Note 3)	Index Register Plus Offset	0	New Operand Data (Note 3)
STS STX	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data (High Order Byte)
		7	1	Index Register Plus Offset + 1	0	Operand Data (Low Order Byte)
JSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer - 2	1	Irrelevant Data (Note 1)
		7	0	Index Register	1	Irrelevant Data (Note 1)
		8	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
EXTENDED						
JMP	3	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Jump Address (High Order Byte)
		3	1	Op Code Address + 2	1	Jump Address (Low Order Byte)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data
CPX LDS LDX	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data (High Order Byte)
		5	1	Address of Operand + 1	1	Operand Data (Low Order Byte)
STA A STA B	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Destination Address (High Order Byte)
		3	1	Op Code Address + 2	1	Destination Address (Low Order Byte)
		4	0	Operand Destination Address	1	Irrelevant Data (Note 1)
		5	1	Operand Destination Address	0	Data from Accumulator
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Current Operand Data
		5	0	Address of Operand	1	Irrelevant Data (Note 1)
		6	1/0 (Note 3)	Address of Operand	0	New Operand Data (Note 3)



## MC6808

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
EXTENDED (Continued)						
STS STX	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	0	Address of Operand	1	Irrelevant Data (Note 1)
		5	1	Address of Operand	0	Operand Data (High Order Byte)
		6	1	Address of Operand + 1	0	Operand Data (Low Order Byte)
JSR	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Subroutine (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
		4	1	Subroutine Starting Address	1	Op Code of Next Instruction
		5	1	Stack Pointer	0	Return Address (Low Order Byte)
		6	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		7	0	Stack Pointer - 2	1	Irrelevant Data (Note 1)
		8	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		9	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
INHERENT						
ABA DAA SEC ASL DEC SEI ASR INC SEV CBA LSR TAB CLC NEG TAP CLI NOP TBA CLR ROL TPA CLV ROR TST COM SBA	2	1	1	Op Code Address	1	Op Code
2		1	Op Code Address + 1	1	Op Code of Next Instruction	
DES DEX INS INX	4	1	1	Op Code Address	1	Op Code
2		1	Op Code Address + 1	1	Op Code of Next Instruction	
3		0	Previous Register Contents	1	Irrelevant Data (Note 1)	
4		0	New Register Contents	1	Irrelevant Data (Note 1)	
PSH	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	1	Stack Pointer	0	Accumulator Data
		4	0	Stack Pointer - 1	1	Accumulator Data
PUL	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Operand Data from Stack
TSX	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	0	New Index Register	1	Irrelevant Data (Note 1)
TXS	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	0	Index Register	1	Irrelevant Data
		4	0	New Stack Pointer	1	Irrelevant Data
RTS	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Address of Next Instruction (High Order Byte)
		5	1	Stack Pointer + 2	1	Address of Next Instruction (Low Order Byte)





MC6808

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INHERENT (Continued)						
WAI	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer - 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer - 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer - 4	0	Contents of Accumulator A
		8	1	Stack Pointer - 5	0	Contents of Accumulator B
		9	1	Stack Pointer - 6 (Note 4)	1	Contents of Cond. Code Register
RTI	10	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Contents of Cond. Code Register from Stack
		5	1	Stack Pointer + 2	1	Contents of Accumulator B from Stack
		6	1	Stack Pointer + 3	1	Contents of Accumulator A from Stack
		7	1	Stack Pointer + 4	1	Index Register from Stack (High Order Byte)
		8	1	Stack Pointer + 5	1	Index Register from Stack (Low Order Byte)
		9	1	Stack Pointer + 6	1	Next Instruction Address from Stack (High Order Byte)
		10	1	Stack Pointer + 7	1	Next Instruction Address from Stack (Low Order Byte)
SWI	12	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 1)
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer - 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer - 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer - 4	0	Contents of Accumulator A
		8	1	Stack Pointer - 5	0	Contents of Accumulator B
		9	1	Stack Pointer - 6	0	Contents of Cond. Code Register
		10	0	Stack Pointer - 7	1	Irrelevant Data (Note 1)
		11	1	Vector Address FFFA (Hex)	1	Address of Subroutine (High Order Byte)
		12	1	Vector Address FFFB (Hex)	1	Address of Subroutine (Low Order Byte)
RELATIVE						
BCC BHI BNE BCS BLE BPL BEQ BLS BRA BGE BLT BVC BGT BMI BVS	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		4	0	Branch Address	1	Irrelevant Data (Note 1)
BSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer - 2	1	Irrelevant Data (Note 1)
		7	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
		8	0	Subroutine Address	1	Irrelevant Data (Note 1, Note 5)

Note 1. If device which is addressed during this cycle uses VMA, then the Data Bus will go to the high impedance three-state condition. Depending on bus capacitance, data from the previous cycle may be retained on the Data Bus.

Note 2. Data is ignored by the MPU.

Note 3. For TST, VMA = 0 and Operand data does not change.

Note 4. While the MPU is waiting for the interrupt, Bus Available will go high, VMA is low.

Note 5. MS Byte = MS Byte of BSR instruction address, LS Byte = LS Byte of subroutine address.

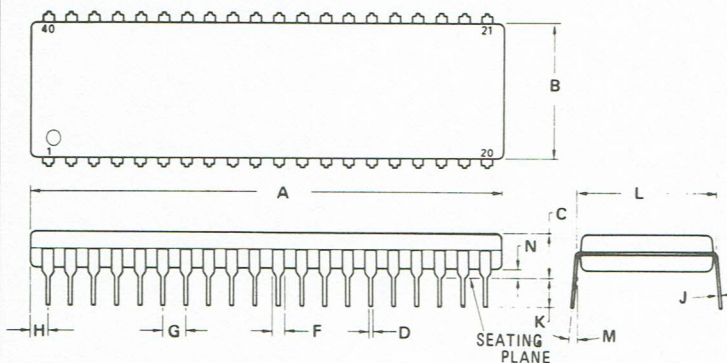


**MOTOROLA Semiconductor Products Inc.**



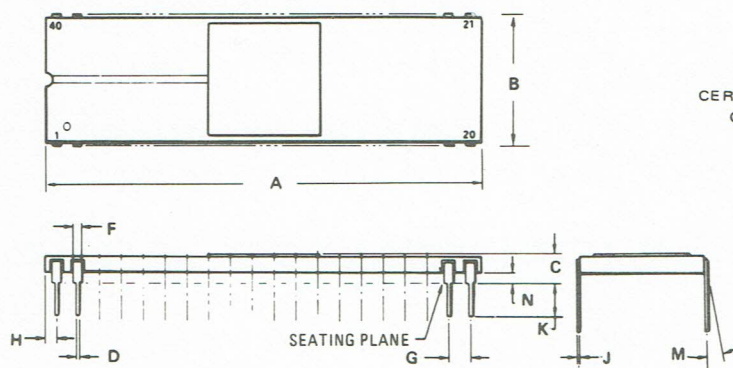
## MC6808

P SUFFIX  
PLASTIC PACKAGE  
CASE 711-03



- NOTES:
1. POSITIONAL TOLERANCE OF LEADS (D), SHALL BE WITHIN 0.25 mm (0.010) AT MAXIMUM MATERIAL CONDITION, IN RELATION TO SEATING PLANE AND EACH OTHER.
  2. DIMENSION L TO CENTER OF LEADS WHEN FORMED PARALLEL.
  3. DIMENSION B DOES NOT INCLUDE MOLD FLASH.
  4. 711-02 OBSOLETE, NEW STANDARD 711-03.

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	51.69	52.45	2.035	2.065
B	13.72	14.22	0.540	0.560
C	3.94	5.08	0.155	0.200
D	0.36	0.56	0.014	0.022
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.16	0.065	0.085
J	0.20	0.38	0.008	0.015
K	2.92	3.43	0.115	0.135
L	15.24 BSC		0.600 BSC	
M	0°	15°	0°	15°
N	0.51	1.02	0.020	0.040



L SUFFIX  
CERAMIC PACKAGE  
CASE 715-02

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
M	0°	10°	0°	10°
N	0.51	1.52	0.020	0.060

- NOTE:
1. LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX. MAT'L CONDITION.







*Appendix C*

**DEFINITION OF THE  
EXECUTABLE INSTRUCTIONS**

## Appendix C

DEFINITION OF THE  
EXERCITABLE INSTRUCTIONS

## A.1 Nomenclature

The following nomenclature is used in the subsequent definitions.

### (a) Operators

( )	= contents of
←	= is transferred to
↑	= "is pulled from stack"
↓	= "is pushed into stack"
.	= Boolean AND
⊕	= Boolean (Inclusive) OR
⊕	= Exclusive OR
≈	= Boolean NOT

### (b) Registers in the MPU

ACCA	= Accumulator A
ACCB	= Accumulator B
ACCX	= Accumulator ACCA or ACCB
CC	= Condition codes register
IX	= Index register, 16 bits
IXH	= Index register, higher order 8 bits
IXL	= Index register, lower order 8 bits
PC	= Program counter, 16 bits
PCH	= Program counter, higher order 8 bits
PCL	= Program counter, lower order 8 bits
SP	= Stack pointer
SPH	= Stack pointer high
SPL	= Stack pointer low

### (c) Memory and Addressing

M	= A memory location (one byte)
M + 1	= The byte of memory at 0001 plus the address of the memory location indicated by "M."
Rel	= Relative address (i.e. the two's complement number stored in the second byte of machine code corresponding to a branch instruction).

### (d) Bits 0 thru 5 of the Condition Codes Register

C	= Carry — borrow	bit — 0
V	= Two's complement overflow indicator	bit — 1
Z	= Zero indicator	bit — 2
N	= Negative indicator	bit — 3
I	= Interrupt mask	bit — 4
H	= Half carry	bit — 5

### (e) Status of Individual Bits BEFORE Execution of an Instruction

An	= Bit n of ACCA (n=7,6,5,...,0)
Bn	= Bit n of ACCB (n=7,6,5,...,0)
IXHn	= Bit n of IXH (n=7,6,5,...,0)

IXLn = Bit n of IXL (n=7,6,5,...,0)

Mn = Bit n of M (n=7,6,5,...,0)

SPHn = Bit n of SPH (n=7,6,5,...,0)

SPLn = Bit n of SPL (n=7,6,5,...,0)

Xn = Bit n of ACCX (n=7,6,5,...,0)

(f) *Status of Individual Bits of the RESULT of Execution of an Instruction*

(i) For 8-bit Results

Rn = Bit n of the result (n = 7,6,5,...,0)

This applies to instructions which provide a result contained in a single byte of memory or in an 8-bit register.

(ii) For 16-bit Results

RHn = Bit n of the more significant byte of the result

(n = 7,6,5,...,0)

RLn = Bit n of the less significant byte of the result

(n = 7,6,5,...,0)

This applies to instructions which provide a result contained in two consecutive bytes of memory or in a 16-bit register.

## A.2 Executable Instructions (definition of)

Detailed definitions of the 72 executable instructions of the source language are provided on the following pages.

**Add Accumulator B to Accumulator A****ABA**

Operation:  $ACCA \leftarrow (ACCA) + (ACCB)$

Description: Adds the contents of ACCB to the contents of ACCA and places the result in ACCA.

Condition Codes: H: Set if there was a carry from bit 3; cleared otherwise.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = A_3 \cdot B_3 + B_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot A_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot B_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot \bar{B}_7 \cdot R_7$$

$$C = A_7 \cdot B_7 + B_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot A_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
Inherent	2	1	1B	033	027



## ADC

**Add with Carry**

Operation:  $ACCX \leftarrow (ACCX) + (M) + (C)$

Description: Adds the contents of the C bit to the sum of the contents of ACCX and M, and places the result in ACCX.

Condition Codes: H Set if there was a carry from bit 3; cleared otherwise.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot X_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot M_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot \bar{M}_7 \cdot R_7$$

$$C = X_7 \cdot M_7 + M_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot X_7$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	89	211	137
A DIR	3	2	99	231	153
A EXT	4	3	B9	271	185
A IND	5	2	A9	251	169
B IMM	2	2	C9	311	201
B DIR	3	2	D9	331	217
B EXT	4	3	F9	371	249
B IND	5	2	E9	351	233

**Add Without Carry****ADD**

Operation:  $ACCX \leftarrow (ACCX) + (M)$

Description: Adds the contents of ACCX and the contents of M and places the result in ACCX.

Condition Codes: H: Set if there was a carry from bit 3; cleared otherwise.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot X_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot M_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot \bar{M}_7 \cdot R_7$$

$$C = X_7 \cdot M_7 + M_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot X_7$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8B	213	139
A DIR	3	2	9B	233	155
A EXT	4	3	BB	273	187
A IND	5	2	AB	253	171
B IMM	2	2	CB	313	203
B DIR	3	2	DB	333	219
B EXT	4	3	FB	373	251
B IND	5	2	EB	353	235

**AND****Logical AND**

Operation:  $ACCX \leftarrow (ACCX) \cdot (M)$

Description: Performs logical "AND" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "AND" of the corresponding bits of M and of ACCX before the operation.)

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

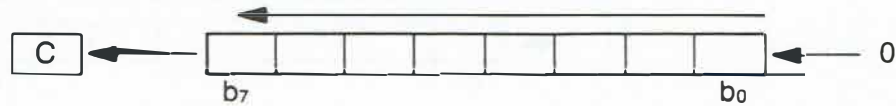
See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	84	204	132
A DIR	3	2	94	224	148
A EXT	4	3	B4	264	180
A IND	5	2	A4	244	164
B IMM	2	2	C4	304	196
B DIR	3	2	D4	324	212
B EXT	4	3	F4	364	244
B IND	5	2	E4	344	228

**Arithmetic Shift Left****ASL**

Operation:



Description: Shifts all bits of the ACCX or M one place to the left. Bit 0 is loaded with a zero. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.  
 C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \vee [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation)

$$C = M_7$$

Addressing Formats

See Table C-3 (Page C-81).

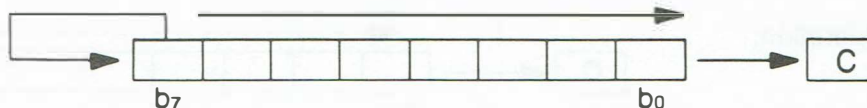
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	48	110	072
B	2	1	58	130	088
EXT	6	3	78	170	120
IND	7	2	68	150	104

# ASR

## Arithmetic Shift Right

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation)

$$C = M_0$$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	47	107	071
B	2	1	57	127	087
EXT	6	3	77	167	119
IND	7	2	67	147	103



**Branch if Carry Clear****BCC**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if (C)=0

Description: Tests the state of the C bit and causes a branch if C is clear.

See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	24	044	036

**BCS****Branch if Carry Set**

Operation:  $PC \leftarrow (PC) + 0002 + \text{Rel if } (C)=1$

Description: Tests the state of the C bit and causes a branch if C is set.  
See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	25	045	037

**Branch if Equal****BEQ**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(Z)=1$

Description: Tests the state of the Z bit and causes a branch if the Z bit is set.  
See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	27	047	039

**BGE****Branch if Greater than or Equal to Zero**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(N) \oplus (V) = 0$

i.e. if  $(ACCX) \geq (M)$   
(Two's complement numbers)

Description: Causes a branch if (N is set and V is set) OR (N is clear and V is clear).

If the BGE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2C	054	044

**Branch if Greater than Zero****BGT**

Operation:  $PC \leftarrow (PC) + 0002 + \text{Rel if } (Z) \odot [(N) \oplus (V)] = 0$   
 i.e. if  $(\text{ACCX}) > (M)$   
 (two's complement numbers)

Description: Causes a branch if [ Z is clear ] AND [(N is set and V is set) OR (N is clear and V is clear)].

If the BGT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2E	056	046



**BHI**

## Branch if Higher

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(C) \cdot (Z) = 0$   
i.e. if  $(ACCX) > (M)$   
(unsigned binary numbers)

**Description:** Causes a branch if (C is clear) AND (Z is clear).

If the BHI instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was greater than the unsigned binary number represented by the subtrahend (i.e. M).

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

### Addressing Formats:

See Table C-8 (Page C-84).

### Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	22	042	034

**Bit Test****BIT**

Operation: (ACCX) · (M)

Description: Performs the logical "AND" comparison of the contents of ACCX and the contents of M and modifies condition codes accordingly. Neither the contents of ACCX or M operands are affected. (Each bit of the result of the "AND" would be the logical "AND" of the corresponding bits of M and ACCX.)

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the result of the "AND" would be set; cleared otherwise.  
 Z: Set if all bits of the result of the "AND" would be cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	85	205	133
A DIR	3	2	95	225	149
A EXT	4	3	B5	265	181
A IND	5	2	A5	245	165
B IMM	2	2	C5	305	197
B DIR	3	2	D5	325	213
B EXT	4	3	F5	365	245
B IND	5	2	E5	345	229

**BLE****Branch if Less than or Equal to Zero**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(Z) \odot [(N) \oplus (V)] = 1$

i.e. if  $(ACCX) \leq (M)$

(two's complement numbers)

Description: Causes a branch if [Z is set] OR [(N is set and V is clear) OR (N is clear and V is set)].

If the BLE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2F	057	047

# BLS

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	23	043	035

BLT

Branch if Less than Zero

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(N) \oplus (V) = 1$   
i.e. if  $(ACCX) < (M)$   
(two's complement numbers)

Description: Causes a branch if (N is set and V is clear) OR (N is clear and V is set).  
If the BLT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than the two's complement number represented by the subtrahend (i.e. M).  
See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2D	055	045



**Branch if Minus****BMI**

Operation:  $PC \leftarrow (PC) + 0002 + \text{Rel if } (N) = 1$

Description: Tests the state of the N bit and causes a branch if N is set.  
See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2B	053	043

BNE

Branch if Not Equal

Operation:           PC ← (PC) + 0002 + Rel if (Z) = 0

Description:        Tests the state of the Z bit and causes a branch if the Z bit is clear.  
                      See BRA instruction for details of the execution of the branch.

Condition Codes:   Not affected.

Addressing Formats:  
See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	26	046	038

**Branch if Plus****BPL**

Operation:  $PC \leftarrow (PC) + 0002 + \text{Rel if } (N) = 0$

Description: Tests the state of the N bit and causes a branch if N is clear.  
See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2A	052	042

## BRA

### Branch Always

Operation:  $PC \leftarrow (PC) + 0002 + Rel$

Description: Unconditional branch to the address given by the foregoing formula, in which R is the relative address stored as a two's complement number in the second byte of machine code corresponding to the branch instruction.

Note: The source program specifies the destination of any branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler obtains the relative address R from the absolute address and the current value of the program counter PC.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	20	040	032

**Branch to Subroutine****BSR**

Operation:         $PC \leftarrow (PC) + 0002$   
                       $\downarrow (PCL)$   
                       $SP \leftarrow (SP) - 0001$   
                       $\downarrow (PCH)$   
                       $SP \leftarrow (SP) - 0001$   
                       $PC \leftarrow (PC) + Rel$

Description:        The program counter is incremented by 2. The less significant byte of the contents of the program counter is pushed into the stack. The stack pointer is then decremented (by 1). The more significant byte of the contents of the program counter is then pushed into the stack. The stack pointer is again decremented (by 1). A branch then occurs to the location specified by the program.

See BRA instruction for details of the execution of the branch.

Condition Codes:   Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	8	2	8D	215	141

**BRANCH TO SUBROUTINE EXAMPLE**

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A.	Before					
	PC	$\leftarrow$ \$1000	8D		BSR	CHARLI
			50			
	SP	$\leftarrow$ \$EFFF				
B.	After					
	PC	$\leftarrow$ \$1052	**	CHARLI	***	*****
	SP	$\leftarrow$ \$EFFF				
		\$EFFF	10			
		\$EFFF	02			



**BVC****Branch if Overflow Clear**

Operation:  $PC \leftarrow (PC) + 0002 + \text{Rel if } (V) = 0$

Description: Tests the state of the V bit and causes a branch if the V bit is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	28	050	040

**Branch if Overflow Set****BVS**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if (V) = 1

Description: Tests the state of the V bit and causes a branch if the V bit is set.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table C-8 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	29	051	041

**CBA****Compare Accumulators**

Operation: (ACCA) – (ACCB)

Description: Compares the contents of ACCA and the contents of ACCB and sets the condition codes, which may be used for arithmetic and logical conditional branches. Both operands are unaffected.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.
- Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.
- V: Set if the subtraction would cause two's complement overflow; cleared otherwise.
- C: Set if the subtraction would require a borrow into the most significant bit of the result; clear otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot \bar{B}_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot B_7 \cdot R_7$$

$$C = \bar{A}_7 \cdot B_7 + B_7 \cdot R_7 + R_7 \cdot \bar{A}_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	11	021	017

**Clear Carry****CLC**

Operation:  $C \text{ bit} \leftarrow 0$

Description: Clears the carry bit in the processor condition codes register.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Cleared

Boolean Formulae for Condition Codes:  
 $C = 0$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0C	014	012

**CLI****Clear Interrupt Mask**

Operation:  $I \text{ bit} \leftarrow 0$

Description: Clears the interrupt mask bit in the processor condition codes register. This enables the microprocessor to service an interrupt from a peripheral device if signalled by a high state of the "Interrupt Request" control input.

Condition Codes: H: Not affected.  
 I: Cleared.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$I = 0$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0E	016	014



**Clear****CLR**

Operation:  $ACCX \leftarrow 00$

or:  $M \leftarrow 00$

Description: The contents of ACCX or M are replaced with zeros.

Condition Codes: H: Not affected.

I: Not affected.

N: Cleared

Z: Set

V: Cleared

C: Cleared

Boolean Formulae for Condition Codes:

$N = 0$

$Z = 1$

$V = 0$

$C = 0$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4F	117	079
B	2	1	5F	137	095
EXT	6	3	7F	177	127
IND	7	2	6F	157	111

**CLV****Clear Two's Complement Overflow Bit**

Operation:  $V \text{ bit} \leftarrow 0$

Description: Clears the two's complement overflow bit in the processor condition codes register.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Not affected.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$V = 0$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0A	012	010

**Compare****CMP**

Operation: (ACCX) – (M)

Description: Compares the contents of ACCX and the contents of M and determines the condition codes, which may be used subsequently for controlling conditional branching. Both operands are unaffected.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.  
 Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.  
 V: Set if the subtraction would cause two's complement overflow; cleared otherwise.  
 C: Carry is set if the absolute value of the contents of memory is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

$$V = X_7 \cdot \overline{M_7} \cdot \overline{R_7} + \overline{X_7} \cdot M_7 \cdot R_7$$

$$C = \overline{X_7} \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \overline{X_7}$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	81	201	129
A DIR	3	2	91	221	145
A EXT	4	3	B1	261	177
A IND	5	2	A1	241	161
B IMM	2	2	C1	301	193
B DIR	3	2	D1	321	209
B EXT	4	3	F1	361	241
B IND	5	2	E1	341	225

**COM****Complement**

Operation:  $ACCX \leftarrow \approx (ACCX) = FF - (ACCX)$

or:  $M \leftarrow \approx (M) = FF - (M)$

Description: Replaces the contents of ACCX or M with its one's complement. (Each bit of the contents of ACCX or M is replaced with the complement of that bit.)

Condition Codes: H: Not affected.

I: Not affected.

N: Set if most significant bit of the result is set; cleared otherwise.

Z: Set if all bits of the result are cleared; cleared otherwise.

V: Cleared.

C: Set.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

$$C = 1$$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	43	103	067
B	2	1	53	123	083
EXT	6	3	73	163	115
IND	7	2	63	143	099

**Compare Index Register****CPX**

Operation:  $(IXL) - (M + 1)$   
 $(IXH) - (M)$

Description: The more significant byte of the contents of the index register is compared with the contents of the byte of memory at the address specified by the program. The less significant byte of the contents of the index register is compared with the contents of the next byte of memory, at one plus the address specified by the program. The Z bit is set or reset according to the results of these comparisons, and may be used subsequently for conditional branching.

The N and V bits, though determined by this operation, are not intended for conditional branching.

The C bit is not affected by this operation.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the result of the subtraction from the more significant byte of the index register would be set; cleared otherwise.  
 Z: Set if all bits of the results of both subtractions would be cleared; cleared otherwise.  
 V: Set if the subtraction from the more significant byte of the index register would cause two's complement overflow; cleared otherwise.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = IXH_7 \cdot \overline{M_7} \cdot \overline{RH_7} + IXH_7 \cdot M_7 \cdot RH_7$$

Addressing Formats:

See Table C-5 (Page C-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8C	214	140
DIR	4	2	9C	234	156
EXT	5	3	BC	274	188
IND	6	2	AC	254	172



**DAA****Decimal Adjust ACCA**

Operation: Adds hexadecimal numbers 00, 06, 60, or 66 to ACCA, and may also set the carry bit, as indicated in the following table:

State of C-bit before DAA (Col. 1)	Upper Half-byte (bits 4-7) (Col. 2)	Initial Half-carry H-bit (Col.3)	Lower to ACCA (bits 0-3) (Col. 4)	Number Added after by DAA (Col. 5)	State of C-bit DAA (Col. 6)
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-3	1	0-3	66	1

Note: Columns (1) through (4) of the above table represent all possible cases which can result from any of the operations ABA, ADD, or ADC, with initial carry either set or clear, applied to two binary-coded-decimal operands. The table shows hexadecimal values.

Description: If the contents of ACCA and the state of the carry-borrow bit C and the half-carry bit H are all the result of applying any of the operations ABA, ADD, or ADC to binary-coded-decimal operands, with or without an initial carry, the DAA operation will function as follows.

Subject to the above condition, the DAA operation will adjust the contents of ACCA and the C bit to represent the correct binary-coded-decimal sum and the correct state of the carry.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Not defined.  
 C: Set or reset according to the same rule as if the DAA and an immediately preceding ABA, ADD, or ADC were replaced by a hypothetical binary-coded-decimal addition.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

C = See table above.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	19	031	025

**DEC****Decrement**

Operation:  $ACCX \leftarrow (ACCX) - 01$

or:  $M \leftarrow (M) - 01$

Description: Subtract one from the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow occurs if and only if (ACCX) or (M) was 80 before the operation.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{X}_6 \cdot \bar{X}_5 \cdot \bar{X}_4 \cdot \bar{X}_3 \cdot \bar{X}_2 \cdot \bar{X}_0 = \bar{R}_7 \cdot R_6 \cdot R_5 \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1 \cdot R_0$$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4A	112	074
B	2	1	5A	132	090
EXT	6	3	7A	172	122
IND	7	2	6A	152	106

**Decrement Stack Pointer****DES**

Operation:  $SP \leftarrow (SP) - 0001$

Description: Subtract one from the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	34	064	052

**DEX****Decrement Index Register**

Operation:  $IX \leftarrow (IX) - 0001$

Description: Subtract one from the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	09	011	009



**Exclusive OR****EOR**

Operation:  $ACCX \leftarrow (ACCX) \oplus (M)$

Description: Perform logical "EXCLUSIVE OR" between the contents of ACCX and the contents of M, and place the result in ACCX. (Each bit of ACCX after the operation will be the logical "EXCLUSIVE OR" of the corresponding bit of M and ACCX before the operation.)

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	88	210	136
A DIR	3	2	98	230	152
A EXT	4	3	B8	270	184
A IND	5	2	A8	250	168
B IMM	2	2	C8	310	200
B DIR	3	2	D8	330	216
B EXT	4	3	F8	370	248
B IND	5	2	E8	350	232

**INC****Increment**

Operation:  $ACCX \leftarrow (ACCX) + 01$

or:  $M \leftarrow (M) + 01$

Description: Add one to the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes: H: Not affected.

I: Not affected.

N: Set if most significant bit of the result is set; cleared otherwise.

Z: Set if all bits of the result are cleared; cleared otherwise.

V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow will occur if and only if (ACCX) or (M) was 7F before the operation.

C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = \bar{X}_7 \cdot X_6 \cdot X_5 \cdot X_4 \cdot X_3 \cdot X_2 \cdot X_1 \cdot X_0$$

$$C = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4C	114	076
B	2	1	5C	134	092
EXT	6	3	7C	174	124
IND	7	2	6C	154	108

**Increment Stack Pointer****INS**Operation:  $SP \leftarrow (SP) + 0001$ 

Description: Add one to the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	31	061	049

**INX****Increment Index Register**

Operation:  $IX \leftarrow (IX) + 0001$

Description: Add one to the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Set if all 16 bits of the result are cleared; cleared otherwise.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	08	010	008

**Jump****JMP**

Operation: PC ← numerical address

Description: A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table C-7 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	3	3	7E	176	126
IND	4	2	6E	156	110



# JSR

## Jump to Subroutine

Operation:

Either:  $PC \leftarrow (PC) + 0003$  (for EXTended addressing)

or:  $PC \leftarrow (PC) + 0002$  (for INDexed addressing)

Then:  $\downarrow (PCL)$

$SP \leftarrow (SP) - 0001$

$\downarrow (PCH)$

$SP \leftarrow (SP) - 0001$

$PC \leftarrow$  numerical address

Description: The program counter is incremented by 3 or by 2, depending on the addressing mode, and is then pushed onto the stack, eight bits at a time. The stack pointer points to the next empty location in the stack. A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table C-7 (Page C-84).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	9	3	BD	275	189
IND	8	2	AD	255	173

### JUMP TO SUBROUTINE EXAMPLE (extended mode)

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. Before:	PC	→ \$0FFF	BD		JSR	CHARLI
		\$1000	20			
		\$1001	77			
	SP	← \$EFFF				
B. After:	PC	→ \$2077	**	CHARLI	***	*****
	SP	→ \$EFFF				
		\$EFFE	10			
		\$EFFF	02			

**Load Accumulator****LDA**

Operation:  $ACCX \leftarrow (M)$

Description: Loads the contents of memory into the accumulator. The condition codes are set according to the data.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	86	206	134
A DIR	3	2	96	226	150
A EXT	4	3	B6	266	182
A IND	5	2	A6	246	166
B IMM	2	2	C6	306	198
B DIR	3	2	D6	326	214
B EXT	4	3	F6	366	246
B IND	5	2	E6	346	230

**LDS****Load Stack Pointer**

Operation:  $SPH \leftarrow (M)$   
 $SPL \leftarrow (M+1)$

Description: Loads the more significant byte of the stack pointer from the byte of memory at the address specified by the program, and loads the less significant byte of the stack pointer from the next byte of memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the stack pointer is set by the operation; cleared otherwise.  
 Z: Set if all bits of the stack pointer are cleared by the operation; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = 0$$

Addressing Formats:

See Table C-5 (Page C-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8E	216	142
DIR	4	2	9E	236	158
EXT	5	3	BE	276	190
IND	6	2	AE	256	174

**Load Index Register****LDX**

Operation:  $IXH \leftarrow (M)$   
 $IXL \leftarrow (M+1)$

Description: Loads the more significant byte of the index register from the byte of memory at the address specified by the program, and loads the less significant byte of the index register from the next byte of memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the index register is set by the operation; cleared otherwise.  
 Z: Set if all bits of the index register are cleared by the operation; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = 0$$

Addressing Formats:

See Table C-5 (Page C-82).

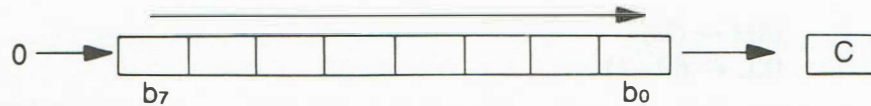
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	CE	316	206
DIR	4	2	DE	336	222
EXT	5	3	FE	376	254
IND	6	2	EE	356	238

## LSR

### Logical Shift Right

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded with a zero. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Cleared.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.  
 C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = 0$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation).

$$C = M_0$$

Addressing Formats:

See Table C-5 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	44	104	068
B	2	1	54	124	084
EXT	6	3	74	164	116
IND	7	2	64	144	100



**Negate****NEG**

Operation:  $ACCX \leftarrow - (ACCX) = 00 - (ACCX)$

or:  $M \leftarrow - (M) = 00 - (M)$

Description: Replaces the contents of ACCX or M with its two's complement. Note that 80 is left unchanged.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there would be two's complement overflow as a result of the implied subtraction from zero; this will occur if and only if the contents of ACCX or M is 80.  
 C: Set if there would be a borrow in the implied subtraction from zero; the C bit will be set in all cases except when the contents of ACCX or M is 00.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = R_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$C = R_7 + R_6 + R_5 + R_4 + R_3 + R_2 + R_1 + R_0$$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	40	100	064
B	2	1	50	120	080
EXT	6	3	70	160	112
IND	7	2	60	140	096

## NOP

**No Operation**

Description: This is a single-word instruction which causes only the program counter to be incremented. No other registers are affected.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	01	001	001

**Inclusive OR****ORA**

Operation:  $ACCX \leftarrow (ACCX) \odot (M)$

Description: Perform logical "OR" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "OR" of the corresponding bits of M and of ACCX before the operation).

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8A	212	138
A DIR	3	2	9A	232	154
A EXT	4	3	BA	272	186
A IND	5	2	AA	252	170
B IMM	2	2	CA	312	202
B DIR	3	2	DA	332	218
B EXT	4	3	FA	372	250
B IND	5	2	EA	352	234

**PSH****Push Data Onto Stack**

Operation:  $\downarrow$  (ACCX)  
 $SP \leftarrow (SP) - 0001$

Description: The contents of ACCX is stored in the stack at the address contained in the stack pointer. The stack pointer is then decremented.

Condition Codes: Not affected.

Addressing Formats:

See Table C-4 (Page C-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	36	066	054
B	4	1	37	067	055

**Pull Data from Stack****PUL**

Operation:  $SP \leftarrow (SP) + 0001$   
 $\uparrow ACCX$

Description: The stack pointer is incremented. The ACCX is then loaded from the stack, from the address which is contained in the stack pointer.

Condition Codes: Not affected.

Addressing Formats:

See Table C-4 (Page C-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

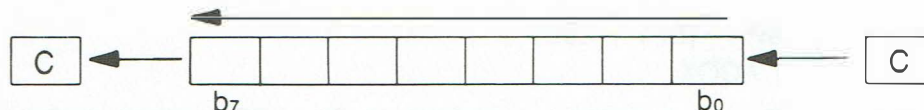
Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	32	062	050
B	4	1	33	063	051



# ROL

Rotate Left

Operation:



Description: Shifts all bits of ACCX or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.  
 C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the rotation)

$$C = M_7$$

Addressing Formats:

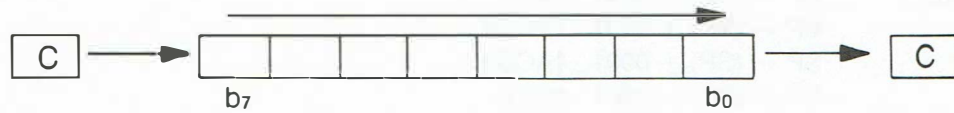
See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	49	111	073
B	2	1	59	131	089
EXT	6	3	79	171	121
IND	7	2	69	151	105

**Rotate Right****ROR**

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded from the C bit. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.  
 C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \cup [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the rotation)

$$C = M_0$$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	46	106	070
B	2	1	56	126	086
EXT	6	3	76	166	118
IND	7	2	66	146	102

**RTI****Return from Interrupt**

Operation:  $SP \leftarrow (SP) + 0001, \uparrow CC$   
 $SP \leftarrow (SP) + 0001, \uparrow ACCB$   
 $SP \leftarrow (SP) + 0001, \uparrow ACCA$   
 $SP \leftarrow (SP) + 0001, \uparrow IXH$   
 $SP \leftarrow (SP) + 0001, \uparrow IXL$   
 $SP \leftarrow (SP) + 0001, \uparrow PCH$   
 $SP \leftarrow (SP) + 0001, \uparrow PCL$

Description: The condition codes, accumulators B and A, the index register, and the program counter, will be restored to a state pulled from the stack. Note that the interrupt mask bit will be reset if and only if the corresponding bit stored in the stack is zero.

Condition Codes: Restored to the states pulled from the stack.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	10	1	3B	073	059

**Return from Interrupt**

Example

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. Before	PC	→ \$D066	3B		RTI	
	SP	→ \$EFF8				
		\$EFF9	11HINZVC	(binary)		
		\$EFFA	12			
		\$EFFB	34			
		\$EFFC	56			
		\$EFFD	78			
		\$EFFE	55			
		\$EFFF	67			
B. After	PC	→ \$5567	**		***	*****
		\$EFF8				
		\$EFF9	11HINZVC	(binary)		
		\$EFFA	12			
		\$EFFB	34			
		\$EFFC	56			
		\$EFFD	78			
		\$EFFE	55			
	SP	→ \$EFFF	67			
CC = HINZVC (binary)						
ACCB = 12 (Hex)			IXH = 56 (Hex)			
ACCA = 34 (Hex)			IXL = 78 (Hex)			

## RTS

## Return from Subroutine

Operation:  $SP \leftarrow (SP) + 0001$   
 $\uparrow PCH$   
 $SP \leftarrow (SP) + 0001$   
 $\uparrow PCL$

Description: The stack pointer is incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of highest significance in the program counter. The stack pointer is again incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of lowest significance in the program counter.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	5	1	39	071	057

## Return from Subroutine

## EXAMPLE

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. Before	PC	\$30A2	39		RTS	
	SP	\$EFFF				
		\$EFFE	10			
		\$EFFF	02			
B. After	PC	\$1002	**		***	*****
		\$EFFF				
		\$EFFE	10			
	SP	\$EFFF	02			

**SBA****Subtract Accumulators**

Operation:  $ACCA \leftarrow (ACCA) - (ACCB)$

Description: Subtracts the contents of ACCB from the contents of ACCA and places the result in ACCA. The contents of ACCB are not affected.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation.  
 C: Carry is set if the absolute value of accumulator B plus previous carry is larger than the absolute value of accumulator A; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot \bar{B}_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot B_7 \cdot R_7$$

$$C = \bar{A}_7 \cdot B_7 + B_7 \cdot R_7 + R_7 \cdot \bar{A}_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	10	020	016



**Subtract with Carry****SBC**

Operation:  $ACCX \leftarrow (ACCX) - (M) - (C)$

Description: Subtracts the contents of M and C from the contents of ACCX and places the result in ACCX.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Carry is set if the absolute value of the contents of memory plus previous carry is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

$$V = X_7 \cdot \overline{M_7} \cdot \overline{R_7} + \overline{X_7} \cdot M_7 \cdot R_7$$

$$C = \overline{X_7} \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \overline{X_7}$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	82	202	130
A DIR	3	2	92	222	146
A EXT	4	3	B2	262	178
A IND	5	2	A2	242	162
B IMM	2	2	C2	302	194
B DIR	3	2	D2	322	210
B EXT	4	3	F2	362	242
B IND	5	2	E2	342	226

SEC

Set Carry

- Operation: C bit ← 1
- Description: Sets the carry bit in the processor condition codes register.
- Condition Codes: H: Not affected.  
I: Not affected.  
N: Not affected.  
Z: Not affected.  
V: Not affected.  
C: Set.

Boolean Formulae for Condition Codes:  
C = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0D	015	013

**Set Interrupt Mask****SEI**

Operation:  $I \text{ bit} \leftarrow 1$

Description: Sets the interrupt mask bit in the processor condition codes register. The microprocessor is inhibited from servicing an interrupt from a peripheral device, and will continue with execution of the instructions of the program, until the interrupt mask bit has been cleared.

Condition Codes: H: Not affected.  
 I: Set.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$I = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0F	017	015

**SEV****Set Two's Complement Overflow Bit**

Operation:  $V \text{ bit} \leftarrow 1$

Description: Sets the two's complement overflow bit in the processor condition codes register.

Condition Codes: H: Not affected.  
I: Not affected.  
N: Not affected.  
Z: Not affected.  
V: Set.  
C: Not affected.

Boolean Formulae for Condition Codes:

$$V = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0B	013	011

**STA****Store Accumulator**

Operation:  $M \leftarrow (ACCX)$

Description: Stores the contents of ACCX in memory. The contents of ACCX remains unchanged.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the contents of ACCX is set; cleared otherwise.  
 Z: Set if all bits of the contents of ACCX are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = X_7$$

$$Z = \overline{X_7} \cdot \overline{X_6} \cdot \overline{X_5} \cdot \overline{X_4} \cdot \overline{X_3} \cdot \overline{X_2} \cdot \overline{X_1} \cdot \overline{X_0}$$

$$V = 0$$

Addressing Formats:

See Table C-2 (Page C-80).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A DIR	4	2	97	227	151
A EXT	5	3	B7	267	183
A IND	6	2	A7	247	167
B DIR	4	2	D7	327	215
B EXT	5	3	F7	367	247
B IND	6	2	E7	347	231



# STS

## Store Stack Pointer

Operation:  $M \leftarrow (SPH)$   
 $M + 1 \leftarrow (SPL)$

Description: Stores the more significant byte of the stack pointer in memory at the address specified by the program, and stores the less significant byte of the stack pointer at the next location in memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the stack pointer is set; cleared otherwise.  
 Z: Set if all bits of the stack pointer are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = SPH_7$$

$$Z = (\overline{SPH_7} \cdot \overline{SPH_6} \cdot \overline{SPH_5} \cdot \overline{SPH_4} \cdot \overline{SPH_3} \cdot \overline{SPH_2} \cdot \overline{SPH_1} \cdot \overline{SPH_0}) \cdot (\overline{SPL_7} \cdot \overline{SPL_6} \cdot \overline{SPL_5} \cdot \overline{SPL_4} \cdot \overline{SPL_3} \cdot \overline{SPL_2} \cdot \overline{SPL_1} \cdot \overline{SPL_0})$$

$$V = 0$$

Addressing Formats:

See Table C-6 (Page C-83).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	9F	237	159
EXT	6	3	BF	277	191
IND	7	2	AF	257	175

**Store Index Register****STX**

Operation:  $M \leftarrow (IXH)$   
 $M + 1 \leftarrow (IXL)$

Description: Stores the more significant byte of the index register in memory at the address specified by the program, and stores the less significant byte of the index register at the next location in memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bite of the index register is set; cleared otherwise.  
 Z: Set if all bits of the index register are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = IXH_7$$

$$Z = (\overline{IXH_7} \cdot \overline{IXH_6} \cdot \overline{IXH_5} \cdot \overline{IXH_4} \cdot \overline{IXH_3} \cdot \overline{IXH_2} \cdot \overline{IXH_1} \cdot \overline{IXH_0}) \cdot (\overline{IXL_7} \cdot \overline{IXL_6} \cdot \overline{IXL_5} \cdot \overline{IXL_4} \cdot \overline{IXL_3} \cdot \overline{IXL_2} \cdot \overline{IXL_1} \cdot \overline{IXL_0})$$

$$V = 0$$

Addressing Formats:

See Table C-6 (Page C-83).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	DF	337	223
EXT	6	3	FF	377	255
IND	7	2	EF	357	239

**SUB****Subtract**

Operation:  $ACCX \leftarrow (ACCX) - (M)$

Description: Subtracts the contents of M from the contents of ACCX and places the result in ACCX.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if the absolute value of the contents of memory are larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{M}_7 \cdot \bar{R}_7 \cdot \bar{X}_7 \cdot M_7 \cdot R_7$$

$$C = \bar{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \bar{X}_7$$

Addressing Formats:

See Table C-1 (Page C-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):  
 (DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	80	200	128
A DIR	3	2	90	220	144
A EXT	4	3	B0	260	176
A IND	5	2	A0	240	160
B IMM	2	2	C0	300	192
B DIR	3	2	D0	320	208
B EXT	4	3	F0	360	240
B IND	5	2	E0	340	224

**Software Interrupt****SWI**

Operation:

$$PC \leftarrow (PC) + 0001$$

$$\downarrow (PCL), SP \leftarrow (SP) - 0001$$

$$\downarrow (PCH), SP \leftarrow (SP) - 0001$$

$$\downarrow (IXL), SP \leftarrow (SP) - 0001$$

$$\downarrow (IXH), SP \leftarrow (SP) - 0001$$

$$\downarrow (ACCA), SP \leftarrow (SP) - 0001$$

$$\downarrow (ACCB), SP \leftarrow (SP) - 0001$$

$$\downarrow (CC), SP \leftarrow (SP) - 0001$$

$$I \leftarrow 1$$

$$PCH \leftarrow (n-0005)$$

$$PCL \leftarrow (n-0004)$$

Description: The program counter is incremented (by 1). The program counter, index register, and accumulator A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

The interrupt mask bit is then set. The program counter is then loaded with the address stored in the software interrupt pointer at memory locations (n-5) and (n-4), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes:

H: Not affected.

I: Set.

N: Not affected.

Z: Not affected.

V: Not affected.

C: Not affected.

Boolean Formula for Condition Codes:

$$I = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	12	1	3F	077	063

**Software Interrupt****EXAMPLE****A. Before:**

CC = HINZVC (binary)

ACCB = 12 (Hex)

ACCA = 34 (Hex)

IXH = 56 (Hex)

IXL = 78 (Hex)

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
PC	→	\$5566	3F		SWI	
SP	→	\$EFFF				
		\$FFFA	D0			
		\$FFFB	55			

**B. After:**

PC → \$D055

SP → \$EFF8

\$EFF9 11HINZVC (binary)

\$EFFA 12

\$EFFB 34

\$EFFC 56

\$EFFD 78

\$EFFE 55

\$EFFF 67

Note: This example assumes that FFFF is the memory location addressed when all lines of the address bus go to the high state.



**TAB****Transfer from Accumulator A to Accumulator B**

Operation:  $ACCB \leftarrow (ACCA)$

Description: Moves the contents of ACCA to ACCB. The former contents of ACCB are lost. The contents of ACCA are not affected.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the contents of the accumulator is set; cleared otherwise.  
 Z: Set if all bits of the contents of the accumulator are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

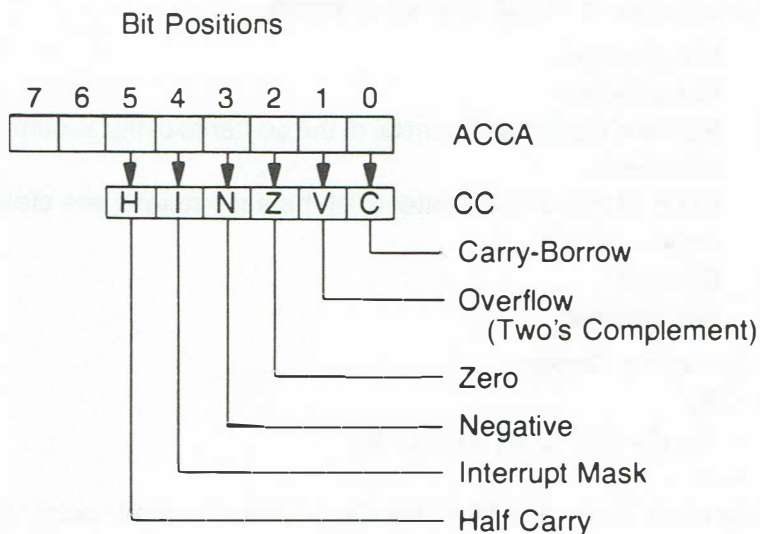
$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	16	026	022

## TAP

Transfer from Accumulator A  
to Processor Condition Codes RegisterOperation:  $CC \leftarrow (ACCA)$ 

Description: Transfers the contents of bit positions 0 thru 5 of accumulator A to the corresponding bit positions of the processor condition codes register. The contents of accumulator A remain unchanged.

Condition Codes: Set or reset according to the contents of the respective bits 0 thru 5 of accumulator A.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	06	006	006

**TBA****Transfer from Accumulator B to Accumulator A**

Operation:  $ACCA \leftarrow (ACCB)$

Description: Moves the contents of ACCB to ACCA. The former contents of ACCA are lost. The contents of ACCB are not affected.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant accumulator bit is set; cleared otherwise.  
 Z: Set if all accumulator bits are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

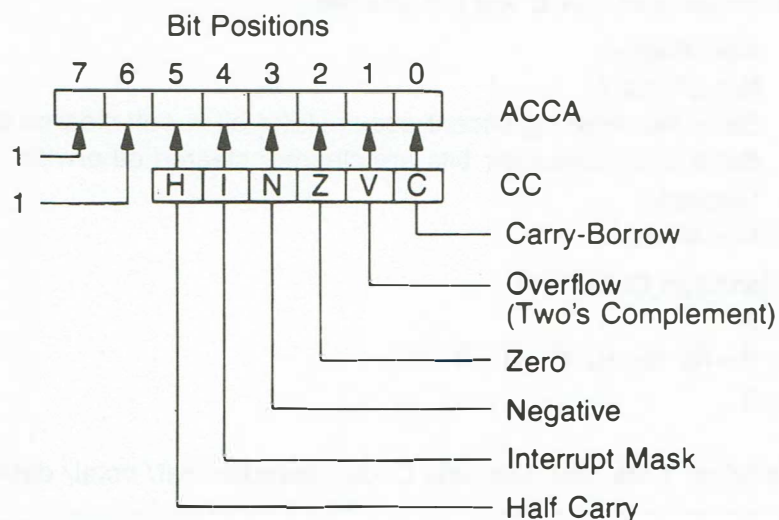
$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	17	027	023

**TPA****Transfer from Processor Condition Codes Register to Accumulator A**Operation:  $ACCA \leftarrow (CC)$ 

**Description:** Transfers the contents of the processor condition codes register to corresponding bit positions 0 thru 5 of accumulator A. Bit positions 6 and 7 of accumulator A are set (i.e. go to the "1" state). The processor condition codes register remains unchanged.

**Condition Codes:** Not affected.

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	07	007	007

**Test****TST**

Operation: (ACCX) – 00  
(M) – 00

Description: Set condition codes N and Z according to the contents of ACCX or M.

Condition Codes: H: Not affected.  
I: Not affected.  
N: Set if most significant bit of the contents of ACCX or M is set; cleared otherwise.  
Z: Set if all bits of the contents of ACCX or M are cleared; cleared otherwise.  
V: Cleared.  
C: Cleared.

Boolean Formulae for Condition Codes:

$$N = M_7$$

$$Z = \overline{M}_7 \cdot \overline{M}_6 \cdot \overline{M}_5 \cdot \overline{M}_4 \cdot \overline{M}_3 \cdot \overline{M}_2 \cdot \overline{M}_1 \cdot \overline{M}_0$$

$$V = 0$$

$$C = 0$$

Addressing Formats:

See Table C-3 (Page C-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4D	115	077
B	2	1	5D	135	093
EXT	6	3	7D	175	125
IND	7	2	6D	155	109



**TSX****Transfer from Stack Pointer to Index Register**

Operation:  $IX \leftarrow (SP) + 0001$

Description: Loads the index register with one plus the contents of the stack pointer. The contents of the stack pointer remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	30	060	048

**Transfer From Index Register to Stack Pointer****TXS**Operation:  $SP \leftarrow (IX) - 0001$ Description: Loads the stack pointer with the contents of the index register, minus one.  
The contents of the index register remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	35	.065	053

**WAI****Wait for Interrupt**

Operation:  $PC \leftarrow (PC) + 0001$   
 $\downarrow (PCL), SP \leftarrow (SP) - 0001$   
 $\downarrow (PCH), SP \leftarrow (SP) - 0001$   
 $\downarrow (IXL), SP \leftarrow (SP) - 0001$   
 $\downarrow (IXH), SP \leftarrow (SP) - 0001$   
 $\downarrow (ACCA), SP \leftarrow (SP) - 0001$   
 $\downarrow (ACCB), SP \leftarrow (SP) - 0001$   
 $\downarrow (CC), SP \leftarrow (SP) - 0001$

Condition Codes: Not affected.

Description: The program counter is incremented (by 1). The program counter, index register, and accumulators A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

Execution of the program is then suspended until an interrupt from a peripheral device is signalled, by the interrupt request control input going to a low state.

When an interrupt is signalled on the interrupt request line, and provided the I bit is clear, execution proceeds as follows. The interrupt mask bit is set. The program counter is then loaded with the address stored in the internal interrupt pointer at memory locations (n-7) and (n-6), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes: H: Not affected.  
 I: Not affected until an interrupt request signal is detected on the interrupt request control line. When the interrupt request is received the I bit is set and further execution takes place, provided the I bit was initially clear.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	9	1	3E	076	062

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
IMMediate	CCC A #number CCC A #symbol CCC A #expression CCC A #'C	CCC B #number CCC B #symbol CCC B #expression CCC B #'C
DIRect or EXTended	CCC A number CCC A symbol CCC A expression	CCC B number CCC B symbol CCC B expression
INDexed	CCC A X CCC Z ,X CCC A number,X CCC A symbol,X CCC A expression,X	CCC B X CCC B ,X CCC B number,X CCC B symbol,X CCC B expression,X

Notes: 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*\*\*".  
 3. "expression" may contain the special symbol "\*\*\*".  
 4. space may be omitted before A or B.

Applicable to the following source instructions:

ADC ADD AND BIT CMP  
 EOR LDA ORA SBC SUB

\*Special symbol indicating program-counter.

**TABLE C-1. Addressing Formats (1)**

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
DIRect or EXTended	STA A number STA A symbol STA A expression	STA B number STA B symbol STA B expression
INDexed	STA A X STA A ,X STA A number,X STA A symbol,X STA A expression,X	STA B X STA B ,X STA B number,X STA B symbol,X STA B expression,X

Notes: 1. "symbol" may be the special symbol "\*\*".  
 2. "expression" may contain the special symbol "\*\*".  
 3. Space may be omitted before A or B.

Applicable to the source instruction:

STA

\*Special symbol indicating program-counter.

**TABLE C-2. Addressing Formats (2)**



Operand or Addressing Mode	Formats
Accumulator A	CCC A
Accumulator B	CCC B
EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*".  
 3. "expression" may contain the special symbol "\*".  
 4. Space may be omitted before A or B.

Applicable to the following source instructions:

ASL ASR CLR COM DEC INC  
 LSR NEG ROL ROR TST

\*Special symbol indicating program-counter.

**TABLE C-3. Addressing Formats (3)**

Addressing Mode	Formats
DIRect or EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.  
2. "symbol" may be the special symbol "\*\*".  
3. "expression" may contain the special symbol "\*\*".

Applicable to the following source instructions:

STS STX

\*Special symbol indicating program-counter.

**TABLE C-6. Addressing Formats (6)**

*Appendix D*

**The PHONEME DICTIONARY**

ABLE	2/A2 1/A2 1/Y 1/B 1/L 85, 45, 69, 4E, 58
ACTIVE	2/AE1 2/EH3 1/K 1/PA0 1/T 1/I2 1/V AF, 80, 59, 43, 6A, 4A, 4F
ADD	1/AE 1/EH3 1/D 6E, 40, 5E
ADDRESS	1/UH2 2/D 2/R 1/EH1 1/S 1/S 71, 9E, AB, 42, 5F, 5F
ADJUST	1/UH2 1/D 2/J 2/UH3 1/UH1 1/S 1/T 71, 5E, 9A, A3, 72, 5F, 6A
ADVANCE	1/EH2 1/D 2/V 1/AE1 1/EH3 1/N 1/S 41, 5E, 8F, 6F, 40, 4D, 5F
AFTER	1/AE 1/F 1/T 1/ER 6E, 5D, 6A, 7A
AGAIN	1/UH1 2/G 1/EH 1/N 72, 9C, 7B, 4D
AIR	1/A2 1/EH2 1/ER 45, 41, 7A
ALL	1/AW 1/UH3 1/L 7D, 63, 58
Am	1/AE1 1/EH3 1/UH3 1/M 6F, 40, 63, 4C
A.M.	1/A2 2/AY 2/Y 2/PA1 2/EH3 1/EH1 1/UH3 1/M 45, A1, A9, BE, 80, 42, 63, 4C
AMERICA	1/UH2 3/M 2/EH2 1/EH2 1/R 1/EH3 1/K 1/UH1 71, CC, 81, 41, 6B, 40, 59, 72
AMOUNT	1/UH2 2/M 2/UH3 1/AH1 1/U1 1/N 1/T 71, 8C, A3, 55, 77, 4D, 6A

An	1/AE1 1/EH3 1/I3 1/N 6F, 40, 49, 4D
AND	1/AE1 1/EH3 1/I3 1/N 1/D 6F, 40, 49, 4D, 5E
ANSWER	2/AE1 1/I3 2/N 1/S 1/R 1/R AF, 49, 8D, 5F, 6B, 6B
APRIL	1/A1 2/AY 1/P 1/R 1/UH3 2/L 46, A1, 65, 6B, 63, 98
ARE	1/AH1 1/UH3 1/ER 55, 63, 7A
ASSIGN	1/UH2 3/S 1/AH1 1/E1 1/N 71, DF, 55, 7C, 4D
ASSIST	1/UH2 0/S 3/S 1/I2 0/S 0/T 71, 1F, DF, 4A, 1F, 2A
AT	1/AE1 1/EH3 1/T 6F, 40, 6A
AUGUST	1/AW 2/G 1/AH2 1/S 1/T 7D, 9C, 48, 5F, 6A
AUTOMATIC	1/AW1 1/T 1/UH2 1/M 2/AE1 2/EH3 1/T 1/I1 1/K 53, 6A, 71, 4C, AF, 80, 6A, 4B, 59
AVERAGE	1/AE1 1/I3 1/V 1/R 1/I2 1/D 1/J 6F, 49, 4F, 6B, 4A, 5E, 5A
AWAY	1/UH2 2/W 1/A1 1/AY 1/AY 71, AD, 46, 61, 61
BACK	3/B 2/AE1 1/EH3 1/K CE, AF, 40, 59
BE	2/B 1/E1 1/E1 8E, 7C, 7C



BEFORE	1/B 1/E1 3/F 1/O 1/R 4E, 7C, DD, 66, 6B
BEGIN	1/B 1/E1 3/G 1/I3 1/I1 1/N 4E, 7C, DC, 49, 4B, 4D
BETWEEN	1/B 1/I2 3/T 1/W 1/E1 1/E1 1/N 4E, 4A, EA, 6D, 7C, 7C, 4D
BREAK	1/B 1/R 1/A1 1/AY 2/K 4E, 6B, 46, 61, 99
BUS	1/B 1/UH1 1/UH3 2/S 4E, 72, 63, 9F
BUT	1/B 1/UH1 1/UH3 2/T 4E, 72, 63, AA
BY	2/B 1/AH1 1/E1 8E, 55, 7C
CALL	2/K 1/AW 1/L 99, 7D, 58
CAN	2/K 1/AE 1/EH3 1/EH3 1/N 99, 6E, 40, 40, 4D
CAR	3/K 1/AH 1/R D9, 64, 6B
CAREFUL	• 2/K 1/A2 2/EH2 2/R 1/F 1/UH3 1/L 99, 45, 81, AB, 5D, 63, 58
CAUTION	2/K 2/AW 1/SH 1/UH3 1/N 99, BD, 51, 63, 4D
CENTS	2/S 2/EH3 1/EH1 1/N 1/T 1/S 9F, 80, 42, 4D, 6A, 5F
CHANGE	2/T 2/CH 2/EH3 1/A1 1/Y 1/N 1/D 1/J AA, 90, 80, 46, 69, 4D, 5E, 5A

CHARACTER 2/K 1/A2 2/EH2 1/R 1/EH1 1/K 1/T 1/ER  
99, 45, 81, 6B, 42, 59, 6A, 7A

CHARGE 2/T 2/CH 1/AH 1/R 1/D 1/J  
AA, 90, 64, 6B, 5E, 5A

CHECK 2/T 2/CH 1/EH 1/K  
AA, 90, 7B, 59

CLASS 2/K 1/L 1/AE 1/UH3 1/S  
99, 58, 6E, 63, 5F

CLEAR 2/K 2/L 2/I3 1/I1 1/R  
99, 98, 89, 4B, 6B

CLOCK 2/K 2/L 1/AH1 1/UH3 1/K  
99, 98, 55, 63, 59

CLOSE to you, 2/K 2/L 1/UH3 1/O1 1/U1 1/S  
99, 98, 63, 75, 77, 5F

CLOSE the door, 2/K 2/L 1/UH3 1/O1 1/U1 1/Z  
99, 98, 63, 75, 77, 52

CODE 2/K 1/O2 1/O1 1/U1 1/D  
99, 74, 75, 77, 5E

COLLECT 1/K 1/UH2 3/L 2/UH3 1/EH2 1/EH2 1/K 1/PA0 1/T  
59, 71, D8, A3, 41, 41, 59, 43, 6A

COMPLETE 1/K 1/UH3 1/M 3/P 1/L 1/E 1/T  
59, 63, 4C, E5, 58, 6C, 6A

CONDITION 1/K 1/UH3 1/N 2/D 1/I2 1/SH 1/UH3 1/N  
59, 63, 4D, 9E, 4A, 51, 63, 4D

CONFIRM 1/K 1/UH3 1/N 3/F 1/R 1/R 1/M  
59, 63, 4D, DD, 6B, 6B, 4C

CONGRATULATIONS 1/K 2/UH3 1/N 1/G 1/R 1/AE1 1/D 2/CH 2/IU 2/L 1/AY 1/SH 1/UH1 2/N 2/Z  
59, A3, 4D, 5C, 6B, 6F, 5E, 90, B6, 98, 61, 51, 72, 8D, 92

CONTACT 2/K 1/AH1 1/N 1/T 1/AE1 1/EH3 1/K 1/PA0 1/T  
99, 55, 4D, 6A, 6F, 40, 59, 43, 6A

CONTINUE	1/K 1/UH1 1/N 1/T 2/I1 2/I3 2/N 1/Y1 1/IU 1/U1 59, 72, 4D, 6A, 8B, 89, 8D, 62, 76, 77
CORRECT	1/K 1/R 2/R 1/EH1 1/K 1/PA0 1/T 59, 6B, AB, 42, 59, 43, 6A
COST	2/K 1/AW 1/S 1/T 99, 7D, 5F, 6A
DAILY	2/D 2/A1 1/Y 1/L 1/Y 9E, 86, 69, 58, 69
DATA	3/D 1/A1 1/AY 1/DT 1/UH2 DE, 46, 61, 44, 71
DATE	3/D 1/A1 1/AY 1/Y1 1/T DE, 46, 61, 62, 6A
DAY	3/D 1/EH3 1/A1 1/AY DE, 40, 46, 61
DECEMBER	1/D 2/Y1 3/S 2/EH2 2/EH2 1/M 1/B 1/R 5E, A2, DF, 81, 81, 4C, 4E, 6B
DEDUCT	1/D 1/I3 3/D 1/UH2 1/UH2 1/K 1/PA0 1/T 5E, 49, DE, 71, 71, 59, 43, 6A
DEPOSIT	1/D 1/I3 3/P 1/AH1 1/UH3 1/Z 1/I2 1/T 5E, 49, E5, 55, 63, 52, 4A, 6A
DETERMINE	1/D 1/I2 3/T 2/R 1/R 1/M 1/I3 1/N 5E, 4A, EA, AB, 6B, 4C, 49, 4D
DEVICE	1/D 1/EH1 1/EH3 1/V 2/UH3 2/AH2 2/Y 2/S 5E, 42, 40, 4F, A3, 88, A9, 9F
DID	2/D 1/I 1/D 9E, 67, 5E
DIFFERENCE	2/D 1/I2 1/F 1/R 1/EH3 1/N 1/S 9E, 4A, 5D, 6B, 40, 4D, 5F
DIRECT	1/D 1/ER 2/EH1 2/K 2/PA0 1/T 5E, 7A, 82, 99, 83, 6A

DIRECTION	2/D 2/ER 1/EH1 1/K 1/SH 1/UH3 1/N 9E, BA, 42, 59, 51, 63, 4D
DISK	1/D 1/I1 1/I3 1/S 1/K 5E, 4B, 49, 5F, 59
DISPLAY	1/D 1/I1 1/S 1/P 1/L 2/A1 2/I3 2/Y 5E, 4B, 5F, 65, 58, 86, 89, A9
DIVIDE	1/D 1/I3 3/V 1/AH1 1/UH3 1/E1 1/D 5E, 49, CF, 55, 63, 7C, 5E
DO	3/D 1/IU 1/U DE, 76, 68
DOLLAR	3/D 2/UH3 1/AH1 1/L 1/UH3 1/R DE, A3, 55, 58, 63, 6B
DOWN	2/D 1/UH3 1/AH1 1/U1 1/N 9E, 63, 55, 77, 4D
DRIVE	1/D 1/R 1/AH1 1/EH3 1/Y 1/V 5E, 6B, 55, 40, 69, 4F
EAST	1/E 1/Y1 1/S 1/T 6C, 62, 5F, 6A
EDIT	1/EH1 1/D 1/I1 1/T 42, 5E, 4B, 6A
EIGHT	1/A1 1/AY 2/Y1 2/T 46, 61, A2, AA
EIGHTEEN	1/A1 1/AY 2/Y1 2/T 2/PA0 1/E1 1/E1 1/N 46, 61, A2, AA, 83, 7C, 7C, 4D
EIGHTY	2/A1 1/AY 1/Y1 1/D 1/Y 86, 61, 62, 5E, 69
ELEVEN	1/UH3 3/L 1/EH1 1/V 1/EH3 1/N 63, D8, 42, 4F, 40, 4D
ELSE	1/EH1 1/EH3 1/L 1/S 42, 40, 58, 5F

ENABLE	1/EH1 1/N 2/A1 1/Y 1/B 1/UH3 1/L 42, 4D, 86, 69, 4E, 63, 58
END	1/EH1 1/EH3 1/N 1/D 42, 40, 4D, 5E
ENTER	1/EH1 1/N 1/T 1/ER 42, 4D, 6A, 7A
EQUALS	1/E1 1/K 1/W 1/IU 1/L 1/S 7C, 59, 6D, 76, 58, 5F
ERASE	1/AY 1/I1 1/R 1/A1 1/AY 1/Y 1/S 61, 4B, 6B, 46, 61, 69, 5F
ERROR	1/EH1 1/ER 1/O1 1/R 42, 7A, 75, 6B
ESCAPE	1/EH1 1/EH3 1/S 1/K 1/A1 1/Y 1/P 42, 40, 5F, 59, 46, 69, 65
EVEN	1/E 1/V 1/EH2 1/N 6C, 4F, 41, 4D
EVER	2/EH1 1/V 1/R 1/R 82, 4F, 6B, 6B
EXECUTE	1/EH1 1/K 1/PA0 1/S 1/EH1 1/K 1/Y1 0/IU 0/U1 0/T 42, 59, 43, 5F, 42, 59, 62, 36, 37, 2A
FACE	3/F 1/A1 1/AY 1/Y1 1/S DD, 46, 61, 62, 5F
FAST	2/F 1/AE 1/EH3 1/S 1/T 9D, 6E, 40, 5F, 6A
FEBRUARY	1/F 2/EH1 1/B 2/R 1/U1 1/EH3 1/I3 1/R 2/E1 5D, 82, 4E, AB, 77, 40, 49, 6B, BC
FEED	2/F 1/E1 1/Y 1/D 9D, 7C, 69, 5E
FEET	2/F 1/E 1/T 9D, 6C, 6A



FIELD	1/F 1/E1 1/Y 1/L 1/D 5D, 7C, 69, 58, 5E
FIFTEEN	1/F 1/I1 2/F 2/T 1/E1 1/E1 1/N 5D, 4B, 9D, AA, 7C, 7C, 4D
FIFTY	2/F 2/I1 1/F 1/T 1/Y 9D, 8B, 5D, 6A, 69
FILE	3/F 1/AH1 1/E1 1/UH3 1/L DD, 55, 7C, 63, 58
FIND	3/F 1/AH1 1/I3 1/E1 1/N 1/D DD, 55, 49, 7C, 4D, 5E
FIVE	3/F 1/UH3 1/AH1 1/E1 1/V DD, 63, 55, 7C, 4F
FIX	1/F 1/I1 1/I3 1/K 1/PA0 1/S 5D, 4B, 49, 59, 43, 5F
FLOPPY	1/F 1/L 1/AH1 1/UH3 1/P 1/Y 5D, 58, 55, 63, 65, 69
FOR, FOUR	2/F 1/O 1/R 9D, 66, 6B
FORM	2/F 1/O 1/R 1/M 9D, 66, 6B, 4C
FORMAT	1/F 1/O2 1/O2 1/R 1/M 0/AE1 0/EH3 0/T 5D, 74, 74, 6B, 4C, 2F, 00, 2A
FORTY	2/F 1/O1 1/R 1/DT 1/Y 9D, 75, 6B, 44, 69
FORWARD	3/F 1/O1 1/R 1/W 1/ER 1/D DD, 75, 6B, 6D, 7A, 5E
FOURTEEN	1/F 1/O 2/R 2/T 2/PA0 1/E1 1/E1 1/N 5D, 66, AB, AA, 83, 7C, 7C, 4D
FOURTH	3/F 1/O 1/R 1/TH DD, 66, 6B, 79
FREE	2/F 1/R 1/E1 1/Y 9D, 6B, 7C, 69

FRIDAY	1/F 2/R 1/AH1 1/E1 1/D 1/A1 1/AY 5D, AB, 55, 7C, 5E, 46, 61
FROM	3/F 1/R 1/UH1 1/M DD, 6B, 72, 4C
FRONT	3/F 1/R 1/UH1 1/N 1/T DD, 6B, 72, 4D, 6A
FUTURE	3/F 2/Y1 1/U1 2/T 1/CH 1/ER DD, A2, 77, AA, 50, 7A
GOSUB	2/G 2/O1 1/U1 2/S 1/UH1 1/UH3 1/B 9C, B5, 77, 9F, 72, 63 4E
GRADE	3/G 1/R 1/A1 1/AY 1/Y1 1/D DC, 6B, 46, 61, 62, 5E
GREAT	2/G 1/R 1/A1 1/E1 1/T 9C, 6B, 46, 7C, 6A
GROUND	3/G 1/R 1/AH1 1/U1 1/N 1/D DC, 6B, 55, 77, 4D, 5E
GUARD	3/G 1/AH1 1/UH3 1/R 1/D DC, 55, 63, 6B, 5E
GUESS	3/G 1/EH 1/S DC, 7B, 5F
HAD	2/H 1/AE 1/EH3 1/D 9B, 6E, 40, 5E
HANDLE	2/H 2/AE1 1/EH3 1/N 1/D 1/UH3 1/L 9B, AF, 40, 4D, 5E, 63, 58
HARD	3/H 1/AH1 1/R 1/D DB, 55, 6B, 5E

HAS	2/H 1/AE1 1/EH3 1/Z 9B, 6F, 40, 52
HAVE	2/H 1/AE 1/UH3 1/V 9B, 6E, 63, 4F
HE	2/H 1/E 9B, 6C
HEAR	2/H 1/E 1/R 9B, 6C, 6B
HEARD	2/H 1/ER 1/R 1/D 9B, 7A, 6B, 5E
HEART	2/H 1/AH1 1/R 1/T 9B, 55, 6B, 6A
HELLO	3/H 1/EH1 1/UH3 1/L 1/UH3 1/O2 2/U1 DB, 42, 63, 58, 63 74, B7
HELP	3/H 1/EH1 1/UH3 1/L 1/P DB, 42, 63, 58, 65
HER	2/H 1/ER 1/R 9B, 7A, 6B
HIGH	2/H 1/AH1 1/E1 9B, 55, 7C
HIM	2/H 1/I1 1/M 9B, 4B, 4C
HIS	2/H 1/I 1/Z 9B, 67, 52
HOW	3/H 1/AH1 1/O2 1/U1 DB, 55, 74, 77
HUNDRED	3/H 1/UH1 1/N 1/D 1/R 1/EH2 1/D DB, 72, 4D, 5E, 6B, 41, 5E
HURRY	2/H 1/ER 1/R 1/Y 9B, 7A, 6B, 69

I	1/AH1 1/EH3/ 1/E1 55, 40, 7C
IF	1/I 1/F 67, 5D
IMPOSSIBLE	1/I2 1/M 3/P 1/AH1 1/S 1/UH3 1/B 1/L 4A, 4C, E5, 55, 5F, 63, 4E, 58
IMPROVEMENT	1/I3 1/M 3/P 1/R 2/U1 1/V 1/M 1/EH3 1/N 1/T 49, 4C, E5, 6B, B7, 4F, 4C, 40, 4D, 6A
IN	1/I 1/N 67, 4D
INCHES	1/I1 1/N 1/T 1/CH 1/EH3 1/Z 4B, 4D, 6A, 50, 40, 52
INCLUDE	1/I3 1/N 3/K 1/L 1/IU 1/U1 1/D 49, 4D, D9, 58, 76, 77, 5E
INCONSISTENT	1/I1 1/N 1/K 1/UH3 2/N 3/S 1/I2 1/S 1/T 1/EH3 1/N 1/T 8B, 4D, 59, 63, 8D, DF, 4A, 5F, 6A, 40, 4D, 6A
INCREASE	1/I2, 1/N 3/K 1/R 1/E 1/S 4A, 4D, D9, 6B, 6C, 5F
INDICATE	2/I2 1/N 1/D 1/I3 1/K 1/A1 1/AY 1/T 8A, 4D, 5E, 49, 59, 46, 61, 6A
INFORMATION	1/I2 1/N 1/F 1/O2 2/R 3/M 1/A1 1/AY 1/SH 1/UH3 1/N •4A, 4D, 5D, 74, AB, CC, 46, 61, 51, 63, 4D
INITIAL	I/I2 2/N 2/I1 1/SH 1/UH3 1/L 4A, 8D, 8B, 51, 63, 58
INPUT	1/I1, 1/I3 1/N 1/P 1/00 1/T 4B 49, 4D, 65, 57, 6A
INQUIRE	1/I3 2/N 3/K 1/W 1/AH2 1/E1 1/R 49, 8D, D9, 6D, 48, 7C, 6B
INSIDE	1/I3 1/N 2/S 1/AH1 1/E1 1/D 49, 4D, 9F, 55, 7C, 5E

INSTRUCTION	1/I2 1/N 3/S 2/T 1/R 1/UH1 1/K 1/SH 0/UH1 0/N 4A, 4D, DF, AA, 6B, 72, 59, 51, 32, 0D
INSUFFICIENT	1/I2 2/N 1/S 1/UH3 2/F 1/I2 1/SH 1/EH3 1/N 1/T 4A, 8D, 5F, 63, 9D, 4A, 51, 40, 4D, 6A
INSURANCE	1/I1 1/N 3/SH 1/R 1/R 1/EH3 1/N 1/S 4B, 4D, D1, 6B, 6B, 40, 4D, 5F
INTEND	1/I2 2/N 3/T 1/EH1 1/I3 1/N 1/D 4A, 8D, EA, 42, 49, 4D, 5E
INTEREST	2/I2 2/N 1/T 1/R 1/EH2 1/S 1/T 8A, 8D, 6A, 6B, 41, 5F, 6A
INTERRUPT	1/I2 2/N 1/T 1/ER 2/UH1 2/P 2/T 4A, 8D, 6A, 7A, B2, A5, AA
INTO	1/I1 1/N 2/T 1/U 4B, 4D, AA, 68
IS	1/I 2/Z 67, 92
IT	1/I1 1/T 4B, 6A
JANUARY	2/D 1/J 2/AE1 1/I3 1/N 2/Y1 1/U1 1/EH3 1/I3 1/R 2/E1 9E, 5A, AF, 49, 4D, A2, 77, 40, 49, 6B, BC
JULY	1/D 1/J 1/IU 3/L 1/AH1 1/E1 5E, 5A, 76, D8, 55, 7C
JUNE	2/D 3/J 1/IU 1/U 1/N 9E, DA, 76, 68 4D
JUST	2/D 2/J 1/UH1 1/S 1/T 9E, 9A, 72, 5F, 6A

Notice that the D sound must precede the J to form the J sound.



KEEP	2/K 1/E1 1/Y 1/P 99, 7C, 69, 65
KEYBOARD	2/K 1/AY 1/Y 1/B 1/O1 1/O2 1/R 1/D 99, 61, 69, 4E, 75, 74, 6B, 5E
KILL	2/K 1/I1 1/I3 1/L 99, 4B, 49, 58
KIND	2/K 1/AH1 1/E1 1/N 1/D 99, 55, 7C, 4D, 5E
KNEW see NEW, KNOW see No, KNOWLEDGE	2/N 1/AH1 1/L 1/EH3 1/D 1/J 8D, 55, 58 40, 5E, 5A
LEAST	2/L 1/E1 1/Y 1/S 1/T 98, 7C, 69, 5F, 6A
LEAVE	2/L 1/E1 1/Y1 1/V 98, 7C, 62, 4F
LEFT	2/L 1/EH2 1/UH3 1/F 1/T 98, 41, 63, 5D, 6A
LENGTH	2/L 1/EH1 1/EH3 1/N 1/G 1/TH 98, 42, 40, 4D, 5C, 79
LIGHT	2/L 1/UH3 1/AH2 1/Y1 1/T 98, 63, 48, 62, 6A
LIMIT	2/L 1/I2 1/M 1/I3 1/T 98, 4A, 4C, 49, 6A
LINE	2/L 1/AH1 1/Y 1/N 98, 55, 69, 4D

LIST	1/L 1/I1 1/S 1/T 58, 4B, 5F, 6A
LOAD	2/L 1/O1 1/U1 1/D 98, 75, 77, 5E
LOCATION	1/L 1/UH3 1/O2 2/K 1/A2 1/Y1 1/SH 1/UH3 1/N 58, 63, 74, 99, 45, 62, 51, 63, 4D
LOCK	1/L 1/AH1 1/UH3 1/K 58, 55, 63, 59
MADE	1/M 2/A1 1/AY 1/Y1 1/D 4C, 86, 61, 62, 5E
MARCH	2/M 1/AH1 1/UH3 1/R 1/T 1/CH 8C, 55, 63, 6B, 6A, 50
MARK	2/M 1/AH1 1/UH3 1/R 1/K 8C, 55, 63, 6B, 59
MAXIMUM	2/M 1/AE1 2/K 1/S 1/EH3 1/M 1/UH3 1/M 8C, 6F, 99, 5F, 40, 4C, 63, 4C
MAY	1/M 2/EH3 1/A1 1/AY 4C, 80, 46, 61
MEASURE	2/M 2/EH1 1/ZH 1/ER 8C, 82, 47, 7A
MEDIUM	2/M 2/E1 1/D 1/E1 1/UH2 1/M 8C, BC, 5E, 7C, 71, 4C
MEMORY	2/M 1/EH1 1/EH3 1/M 1/O2 1/O2 1/R 1/Y 8C, 42, 40, 4C, 74, 74, 6B, 69
MEN	2/M 1/EH1 1/I3 1/N 8C, 42, 49, 4D
MERGE	2/M 1/ER 1/R 1/D 1/J 8C, 7A, 6B, 5E, 5A

MICRO	2/M 2/UH3 2/AH2 2/AY 2/K 1/R 1/O1 1/U1 8C, A3, 88, A1, 99, 6B, 75, 77
MID	1/M 1/I1 1/I3 1/D 4C, 4B, 49, 5E
MILE	2/M 1/AH1 1/I3 1/UH3 1/L 8C, 55, 49, 63, 58
MILLION	1/M 2/I1 1/L 1/Y1 1/UH3 1/N 4C, 8B, 58, 62, 63, 4D
MINUTES	2/M 1/I1 1/N 1/I3 1/T 1/S 8C, 4B, 4D, 49, 6A, 5F
MISTAKE	1/M 1/I3 2/S 2/T 1/A1 1/AY 1/Y1 1/K 4C, 49, 9F, AA, 46, 61, 62, 59
MONDAY	2/M 1/UH1 1/N 1/D 1/A1 1/AY 8C, 72, 4D, 5E, 46, 61
MONTH	2/M 1/UH1 2/N 1/TH 8C, 72, 8D, 79
Mr.	2/M 2/I2 1/S 1/T 1/ER 8C, 8A, 5F, 6A, 7A
Mrs.	1/M 2/I1 1/S 1/I2 1/Z 4C, 8B, 5F, 4A, 52
MUCH	2/M 1/UH1 1/T 1/CH 8C, 72, 6A, 50
NAME	1/N 2/A1 1/AY 1/Y 1/M 4D, 86, 61, 69, 4C
NATION	1/N 2/A1 1/AY 0/SH 1/EH2 1/N 4D, 86, 61, 11, 41, 4D
NEVER	1/N 2/EH1 1/V 1/ER 4D, 82, 4F, 7A

NEW	1/N 1/IU 1/U 4D, 76, 68
NEXT	1/N 1/EH1 1/EH3 1/K 1/PA0 1/S 1/T 4D, 42, 40, 59, 43, 5F, 6A
NINE	2/N 1/AH1 1/I3 1/Y 1/N 8D, 55, 49, 69, 4D
NO	1/N 1/UH3 1/O1 1/U1 4D, 63, 75, 77
NOON	1/N 1/IU 1/U1 1/U 1/N 4D, 76, 77, 68, 4D
NORTH	2/N 1/O 1/R 1/TH 8D, 66, 6B, 79
NOT	1/N 2/AH1 1/T 4D, 95, 6A
NOVEMBER	1/N 1/O1 3/V 2/EH2 1/EH3 1/M 1/B 1/ER 4D, 75, CF, 81, 40, 4C, 4E, 7A
NOW	2/N 2/UH3 1/AH1 1/O2 1/U1 8D, A3, 55, 74, 77
NUMBER	1/N 2/UH1 1/M 1/B 1/ER 4D, B2, 4C, 4E, 7A
O'CLOCK	1/O1 3/K 1/L 1/AH1 1/K 75, D9, 58, 55, 59
OCTOBER	1/AH1 1/K 2/PA0 3/T 1/O2 1/U1 1/B 1/R 55, 59, 83, EA, 74, 77, 4E, 6B
ON	1/AH 1/UH3 1/N 1/N 64, 63, 4D, 4D
ONCE	2/W 1/UH1 1/UH3 1/N 1/T 1/S AD, 72, 63, 4D, 6A, 5F

ONE	2/W 1/UH 1/N AD, 73, 4D
OPEN	2/O1 1/P 1/I1 1/N B5, 65, 4B, 4D
OPTIONS	2/AH1 1/P 1/SH 1/UH3 1/N 1/Z 95, 65, 51, 63, 4D, 52
OR	2/O1 1/O2 1/R B5, 74, 6B
OUT	1/UH3 1/AH2 1/U1 1/T 63, 48, 77, 6A
OUR	1/AH1 1/U1 1/ER 1/Z 55, 77, 7A, 52
OVER	2/O1 1/U1 1/V 1/ER B5, 77, 4F, 7A
PACIFIC	1/P 2/EH2 3/S 1/I1 1/F 1/I3 1/K 65, 81, DF, 4B, 5D, 49, 59
PAGE	1/P 2/A1 1/AY 1/AY 1/D 1/J 65, 86, 61, 61, 5E, 5A
PAPER	1/P 2/A1 1/AY 1/P 1/ER 65, 86, 61, 65, 7A
PARALLEL	2/P 2/EH1 1/R 2/UH3 1/L 1/EH3 1/UH3 1/L A5, 82, 6B, A3, 58, 40, 63, 58
PARTS	2/P 1/AH1 1/R 1/T 1/S A5, 55, 6B, 6A, 5F
PASS	2/P 1/AE 1/EH3 1/S A5, 6E, 40, 5F
PATTERN	2/P 1/AE1 1/EH3 1/DT 1/ER 1/N A5, 6F, 40, 44, 7A, 4D



PAY	2/P 1/EH3 1/A1 1/AY A5, 40, 46, 61
PEEK	2/P 1/E1 1/Y 1/K A5, 7C, 69, 59
PERCENT	1/P 1/R 3/S 1/EH1 1/N 1/T 65, 6B, DF, 42, 4D, 6A
PERIOD	2/P 2/I2 1/R 1/E1 1/UH3 1/D A5, 8A, 6B, 7C, 63, 5E
PHONE	2/F 1/O 1/U1 1/N 9D, 66, 77, 4D
PLEASE	2/P 2/L 1/E1 1/E1 1/Z A5, 98, 7C, 7C, 52
P.M.	1/P 1/E1 2/E1 2/PA0 2/EH1 1/EH3 1/UH3 1/M 65, 7C, BC, 83, 82, 40, 63, 4C
POINT	1/P 1/O1 1/I3 1/AY 2/N 2/T 65, 75, 49, 61, 8D, AA
POKE	1/P 1/O1 1/U1 1/K 65, 75, 77, 59
POSSIBLE	2/P 1/AH1 1/S 1/UH3 1/B 1/L A5, 55, 5F, 63, 4E, 58
POSITION	1/P 2/UH3 2/Z 1/I1 1/SH 1/UH3 1/N 65, A3, 92, 4B, 51, 63, 4D
PRACTICE	1/P 2/R 1/AE1 2/K 1/PA0 1/T 1/I3 1/S 65, AB, 6F, 99, 43, 6A, 49, 5F
PREPARE	2/P 1/R 1/EH3 3/P 1/EH3 1/A1 1/R A5, 6B, 40, E5, 40, 46, 6B
PRINT	2/P 1/R 1/I1 1/N 1/T A5, 6B, 4B, 4D, 6A
PROBLEM	2/P 2/R 1/AH1 1/B 1/L 1/EH3 1/M A5, AB, 55, 4E, 58, 40, 4C

PROGRAM	1/P 2/R 1/O1 1/G 1/R 1/AE1 1/EH3 1/M 65, AB, 75, 5C, 6B, 6F, 40, 4C
PUT	1/P 1/OO1 1/OO1 1/T 65, 56, 56, 6A
QUEEN	1/K 1/W 1/E1 2/E1 2/N 59, 6D, 7C, BC, 8D
QUICK	1/K 1/W 2/I 2/K 59, 6D, A7, 99
RANDOM	2/R 1/AE1 1/EH3 1/N 1/D 1/UH1 1/M AB, 6F, 40, 4D, 5E, 72, 4C
RED	2/R 1/EH3 1/I3 1/D AB, 40, 49, 5E
READ reed,	2/R 1/E1 1/E1 1/D AB, 7C, 7C, 5E
REGISTER	2/R 1/EH2 1/D 1/J 1/I3 1/S 1/T 1/R AB, 41, 5E, 5A, 49, 5F, 6A, 6B
REMARK	1/R 1/E1 3/M 1/AH1 1/R 1/K 6B, 7C, CC, 55, 6B, 59
REPEAT	1/R 1/E1 3/P 1/E1 1/Y1 1/T 6B, 7C, E5, 7C, 62, 6A
REPLACE	1/R 1/E1 3/P 1/L 1/A1 1/AY 1/S 6B, 7C, E5, 58, 46, 61, 5F
REPORT	1/R 1/E1 3/P 1/O1 1/R 1/T 6B, 7C, E5, 75, 6B, 6A
REQUEST	1/R 1/I3 3/K 1/W 1/EH1 1/S 1/T 6B, 49, D9, 6D, 42, 5F, 6A

RESET	1/R 1/E1 1/S 1/EH1 1/EH3 1/T 6B, 7C, 5F, 42, 40, 6A
RESTORE	1/R 1/E1 1/S 1/T 1/O2 1/O2 1/R 6B, 7C, 5F, 6A, 74, 74, 6B
RESUME	2/R 1/E1 1/Z 1/IU 1/U1 1/U1 1/M AB, 7C, 52, 76, 77, 77, 4C
RETURN	1/R 1/E1 3/T 1/ER 1/R 1/N 6B, 7C, EA, 7A, 6B, 4D
RIGHT	2/R 1/AH2 1/UH3 1/E1 1/T AB, 48, 63, 7C, 6A
ROTATION	1/R 1/O2 3/T 1/A1 1/AY 1/SH 1/UH3 1/N 6B, 74, EA, 46, 61, 51, 63, 4D
SAID	3/S 1/EH1 1/I3 1/D DF, 42, 49, 5E
SAME	2/S 1/EH3 1/A1 0/Y 1/Y1 1/M 9F, 40, 46, 29, 62, 4C
SATURDAY	2/S 1/AE1 2/EH3 1/DT 1/R 1/D 1/A1 1/AY 9F, 6F, 80, 44, 6B, 5E, 46, 61
SAVE	2/S 1/EH3 1/A1 1/AY 1/Y1 1/V 9F, 40, 46, 61, 62, 4F
SAY	2/S 1/EH3 1/A1 1/AY 9F, 40, 46, 61
SCHOOL	2/S 2/K 1/U1 1/U1 1/L 9F, 99, 77, 77, 58
SCREEN	2/S 2/K 2/R 1/E1 1/E1 1/N 9F, 99, AB, 7C, 7C, 4D
SEARCH	2/S 1/ER 1/R 1/T 1/CH 9F, 7A, 6B, 6A, 50
SECOND	1/S 2/EH1 1/K 1/UH2 1/N 1/T 5F, 82, 59, 71, 4D, 6A

SECTION	1/S 2/EH1 2/K 1/SH 1/UH3 1/N 5F, 82, 99, 51, 63, 4D
SEE	2/S 1/E1 1/E1 9F, 7C, 7C
SELECT	1/S 2/UH3 2/L 1/UH3 1/EH1 1/K 1/T 5F, A3, 98, 63, 42, 59, 6A
SELF	2/S 1/EH1 1/UH3 1/L 1/F 9F, 42, 63, 58, 5D
SEND	2/S 1/EH1 1/I3 1/N 1/D 9F, 42, 49, 4D, 5E
SENT	2/S 2/EH3 1/EH1 1/N 1/T 9F, 80, 42, 4D, 6A
SEPARATE	2/S 1/EH1 1/P 1/R 1/EH3 1/T 9F, 42, 65, 6B, 40, 6A
SEPTEMBER	1/S 1/EH2 2/P 3/T 1/EH1 1/M 1/B 1/R 5F, 41, A5, EA, 42, 4C, 4E, 6B
SERVE	1/S 2/ER 1/R 1/V 5F, BA, 6B, 4F
SET	2/S 1/EH1 1/EH3 1/T 9F, 42, 40, 6A
SEVEN	2/S 1/EH1 1/V 1/EH3 1/N 9F, 42, 4F, 40, 4D
SEVENTEEN	2/S 1/EH1 1/V 1/EH3 2/N 2/T 1/E1 1/E1 1/N 9F, 42, 4F, 40, 8D, AA, 7C, 7C, 4D
SEVENTY	2/S 1/EH1 1/V 1/EH3 1/N 1/D 1/Y 9F, 42, 4F, 40, 4D, 5E, 69
SHE	2/SH 1/E1 1/E1 91, 7C, 7C
SHIFT	2/SH 2/I1 1/I3 1/F 1/T 91, 8B, 49, 5D, 6A
SHORT	2/SH 1/O1 1/R 1/T 91, 75, 6B, 6A

SHOULD	2/SH 1/IU 1/IU 1/IU 1/D 91, 76, 76, 76, 5E
SIDE	2/S 1/AH1 1/E1 1/D 9F, 55, 7C, 5E
SIGN	2/S 1/AH1 1/I3 1/E1 1/N 9F, 55, 49, 7C, 4D
SIGNAL	2/S 2/I1 1/G 1/N 1/UH3 1/L 9F, 8B, 5C, 4D, 63, 58
SINCE	2/S 1/I 1/N 1/S 9F, 67, 4D, 5F
SINGLE	2/S 2/I1 1/NG 1/G 1/L 9F, 8B, 54, 5C, 58
SIX	2/S 1/I1 1/K 1/S 9F, 4B, 59, 5F
SIXTEEN	1/S 1/I1 2/K 2/S 2/T 1/E1 1/E1 1/N 5F, 4B, 99, 9F, AA, 7C, 7C, 4D
SIXTY	2/S 1/I1 1/K 1/S 1/T 1/Y 9F, 4B, 59, 5F, 6A, 69
SIZE	2/S 1/AH1 1/I3 1/E1 1/Z 9F, 55, 49, 7C, 52
SLOW	2/S 2/L 1/UH3 1/O1 1/U1 9F, 98, 63, 75, 77
SORT	2/S 1/O 1/R 1/T 9F, 66, 6B, 6A
SOUTH	2/S 1/AH1 1/O2 1/U1 1/TH 9F, 55, 74, 77, 79
SPEED	2/S 2/P 1/E1 1/E1 1/D 9F, A5, 7C, 7C, 5E
SPEND	2/S 2/P 1/EH1 1/I3 1/N 1/D 9F, A5, 42, 49, 4D, 5E
SPRING	2/S 2/P 1/R 1/I1 1/NG 9F, A5, 6B, 4B, 54



SQUARE	1/S 2/K 2/W 1/EH3 1/I3 1/R 5F, 99, AD, 40, 49, 6B
STANDARD	2/S 3/T 1/AE1 1/EH3 1/N 1/D 1/ER 1/D 9F, EA, 6F, 40, 4D, 5E, 7A, 5E
STATE	2/S 2/T 1/A1 1/AY 1/Y1 1/T 9F, AA, 46, 61, 62, 6A
STATEMENT	2/S 3/T 1/A1 1/AY 2/T 1/M 1/EH3 1/N 1/T 9F, EA, 46, 61, AA, 4C, 40, 4D, 6A
STATION	2/S 1/T 2/A1 1/AY 1/SH 1/UH3 1/N 9F, 6A, 86, 61, 51, 63, 4D
STEP	2/S 2/T 1/EH1 1/EH3 1/P 9F, AA, 42, 40, 65
STOP	2/S 2/T 1/AH 1/UH3 1/P 9F, AA, 64, 63, 65
STORE	2/S 2/T 1/O 1/R 9F, AA, 66, 6B
STRING	2/S 2/T 1/R 1/I1 1/I3 1/NG 9F, AA, 6B, 4B, 49, 54
SUBSTITUTE	2/S 1/UH1 1/B 2/S 1/T 1/EH3 1/T 1/IU 1/U1 1/T 9F, 72, 4E, 9F, 6A, 40, 6A, 76, 77, 6A
SUCCESS	1/S 1/UH2 2/K 2/S 1/EH1 1/S 5F, 71, 99, 9F, 42, 5F
SUCH	2/S 1/UH 1/T 1/CH *9F, 73, 6A, 50
SUGGEST	1/S 1/UH2 2/G 2/D 2/J 1/EH 1/S 1/T 5F, 71, 9C, 9E, 9A, 7B, 5F, 6A
SUNDAY	2/S 2/UH1 1/N 2/N 1/D 1/A2 1/AY 9F, B2, 4D, 8D, 5E, 45, 61
SUPPLY	1/S 1/UH2 2/P 2/L 1/AH1 1/E1 5F, 71, A5, 98, 55, 7C
SYNTHESIS	2/S 1/I 1/N 1/TH 2/I1 1/S 1/I1 1/S 9F, 67, 4D, 79, 8B, 5F, 4B, 5F

SYSTEM	2/S 2/I1 2/S 1/T 1/EH1 1/M 9F, 8B, 9F, 6A, 42, 4C
TAB	2/T 2/AE1 2/EH3 2/B AA, AF, 80, 8E
TAUGHT	3/T 1/AW 1/T EA, 7D, 6A
TEACHER	2/T 1/E1 1/T 1/CH 1/ER AA, 7C, 6A, 50, 7A
TECHNICAL	1/T 2/EH1 1/K 1/N 1/I3 1/K 1/UH3 1/L 6A, 82, 59, 4D, 49, 59, 63, 58
TELEPHONE	2/T 2/EH1 1/L 2/UH3 1/F 1/O1 1/U1 1/N AA, 82, 58, A3, 5D, 75, 77, 4D
TELEVISION	2/T 2/EH1 1/L 2/UH3 1/V 1/I2 1/ZH 1/UH3 1/N AA, 82, 58, A3, 4F, 4A, 47, 63, 4D
TEN	2/T 2/EH3 1/EH1 1/N AA, 80, 42, 4D
TERMINATE	2/T 2/ER 1/M 1/I3 1/N 1/A2 1/Y1 1/T AA, BA, 4C, 49, 4D, 45, 62, 6A
TEST	2/T 1/EH 1/S 1/T AA, 7B, 5F, 6A
THAT	2/THV 1/AE1 1/EH3 1/T B8, 6F, 40, 6A
THE	1/THV 1/UH3 2/UH3 3/UH3 78, 63, A3, E3
THEM	2/THV 1/EH1 1/EH3 1/M B8, 42, 40, 4C
THEN	2/THV 1/EH1 1/EH3 1/N B8, 42, 40, 4D
THERE	2/THV 1/EH2 1/A2 1/R B8, 41, 45, 6B

THIRD	2/TH 1/ER 1/R 1/D B9, 7A, 6B, 5E
THIRTEEN	3/TH 1/R 2/ER 2/T 2/PA0 1/E1 1/E1 1/N F9, 6B, BA, AA, 83, 7C, 7C, 4D
THIS	1/THV 2/I 1/S 78, A7, 5F
THREE	2/TH 2/R 1/E1 1/Y B9, AB, 7C, 69
THURSDAY	2/TH 2/TH 2/ER 1/R 2/Z 1/D 1/A1 1/AY B9, B9, BA, 6B, 92, 5E, 46, 61
TIME	2/T 1/AH1 1/E1 1/M AA, 55, 7C, 4C
TO	2/T 1/IU 1/U AA, 76, 68
TODAY	2/T 2/U 1/D 1/A1 1/AY AA, A8, 5E, 46, 61
TRACE	2/T 2/R 1/A1 1/AY 1/Y 1/S AA, AB, 46, 61, 69, 5F
TRANSFER	2/T 2/R 1/AE1 2/I3 1/N 2/S 1/F 1/ER AA, AB, 6F, 89, 4D, 9F, 5D, 7A
TUESDAY	2/T 2/IU 2/U1 1/U1 2/Z 1/D 1/A1 1/AY AA, B6, B7, 77, 92, 5E, 46, 61
UNDERSTAND	1/UH2 2/N 1/D 1/R 2/S 2/T 1/AE1 1/EH3 1/N 1/D 71, 8D, 5E, 6B, 9F, AA, 6F, 40, 4D, 5E
UNITED	1/Y1 1/IU 1/U1 2/N 1/AH1 1/Y1 1/T 1/I3 1/D 62, 76, 77, 8D, 55, 62, 6A, 49, 5E
UNKNOWN	1/UH2 1/N 3/N 1/UH3 1/O1 1/U1 1/N 71, 4D, CD, 63, 75, 77, 4D

UNLIMITED	1/UH2 2/N 2/L 2/I2 1/M 1/I3 1/T 1/I3 1/D 71, 8D, 98, 8A, 4C, 49, 6A, 49, 5E
UNTIL	1/UH2 1/N 2/T 2/I1 1/UH3 1/L 71, 4D, AA, 8B, 63, 58
UPON	1/UH1 3/P 1/AH1 1/UH3 1/N 72, E5, 55, 63, 4D
URGENT	2/R 1/R 1/D 1/J 1/EH2 1/N 1/T AB, 6B, 5E, 5A, 41, 4D, 6A
USE	2/Y1 1/IU 1/U 1/S A2, 76, 68, 5F
USER	2/Y1 1/IU 1/U 1/Z 1/ER A2, 76, 68, 52, 7A
USUAL	2/Y1 1/IU 2/U1 1/ZH 1/IU 1/U1 1/UH3 1/L A2, 76, B7, 47, 76, 77, 63, 58
VALID	2/V 1/AE1 1/UH3 1/L 1/UH3 1/D 8F, 6F, 63, 58, 63, 5E
VALUE	2/V 1/AE1 1/EH3 1/L 1/Y1 1/IU 1/U1 8F, 6F, 40, 58, 62, 76, 77
VERIFY	2/V 1/EH1 1/R 1/EH3 1/F 1/AH1 1/E1 8F, 42, 6B, 40, 5D, 55, 7C
VERY	2/V 1/EH, 1/R 1/Y 8F, 7B, 6B, 69
VOICE	2/V 1/O1 1/UH3 1/E1 1/S 8F, 75, 63, 7C, 5F
VOID	2/V 1/O1 1/UH3 1/E1 1/D 8F, 75, 63, 7C, 5E
VOLUME	2/V 1/AH1 1/L 1/Y1 1/U 1/M 8F, 55, 58, 62, 68, 4C

VOTRAX	2/V 1/O 1/T 1/R 1/AE1 1/EH3 1/K 1/S 8F, 66, 6A, 6B, 6F, 40, 59, 5F
WALK	2/W 1/AW 1/K AD, 7D, 59
WANT	2/W 1/AH1 1/UH3 1/N 1/T AD, 55, 63, 4D, 6A
WATCH	2/W 1/AW2 1/AH1 1/T 1/CH AD, 70, 55, 6A, 50
WEDNESDAY	2/W 1/EH1 2/N 1/Z 1/D 1/A1 1/AY AD, 42, 8D, 52, 5E, 46, 61
WEIGH	2/W 1/EH3 1/A2 1/AY 1/Y1 AD, 40, 45, 61, 62
WELCOME	2/W 1/EH1 2/L 1/K 1/UH1 1/M AD, 42, 98, 59, 72, 4C
WELL	2/W 1/EH1 1/UH3 1/L AD, 42, 63, 58
WENT	2/W 1/EH1 1/N 1/T AD, 42, 4D, 6A
WERE	2/W 1/ER 1/R AD, 7A, 6B
WEST	1/W 2/EH1 1/S 1/T 6D, 82, 5F, 6A
WHAT	1/W 2/UH1 1/T 6D, B2, 6A
WHEN	1/W 2/EH1 1/N 6D, 82, 4D
WHERE	1/W 2/EH2 1/A2 1/R 6D, 81, 45, 6B
WHICH	2/W 2/I1 1/T 1/CH AD, 8B, 6A, 50



WHITE	1/W 2/UH3 1/AH2 1/E1 1/T 6D, A3, 48, 7C, 6A
WHY	2/W 2/UH3 1/AH1 1/E1 AD, A3, 55, 7C
WILL	2/W 1/I2 1/UH3 1/L AD, 4A, 63, 58
WIND (short i)	2/W 1/I1 1/N 1/D AD, 4B, 4D, 5E
WIND (long i)	2/W 1/AH1 1/E1 1/N 1/D AD, 55, 7C, 4D, 5E
WITH	2/W 1/I2 2/TH AD, 4A, B9
WORK	1/W 1/ER 2/R 2/K 6D, 7A, AB, 99
WOULD	2/W 1/U1 1/IU 1/IU 1/D AD, 77, 76, 76, 5E
X/RAY	2/EH3 2/K 1/S 1/R 1/AY 1/AY 80, 99, 5F, 6B, 61, 61
YES	2/Y 1/EH1 1/EH3 1/S A9, 42, 40, 5F
YESTERDAY	2/Y 2/EH1 2/S 1/T 1/R 1/D 1/A1 1/AY A9, 82, 9F, 6A, 6B, 5E, 46, 61
YOU	2/Y 1/IU 1/U A9, 76, 68
YOUR	1/Y 1/O 1/R 69, 66, 6B

ZERO

2/Z 1/I1 1/R 1/UH3 1/O1  
92, 4B, 6B, 63, 75

ZONE

2/Z 2/UH3 1/O1 1/U1 1/N  
92, A3, 75, 77, 4D